With Cheat Sheet !

PYTHON'S COMPANION

THE MOST COMPLETE STEP BY STEP GUIDE TO PYTHON PROGRAMMING

From Best Selling Author

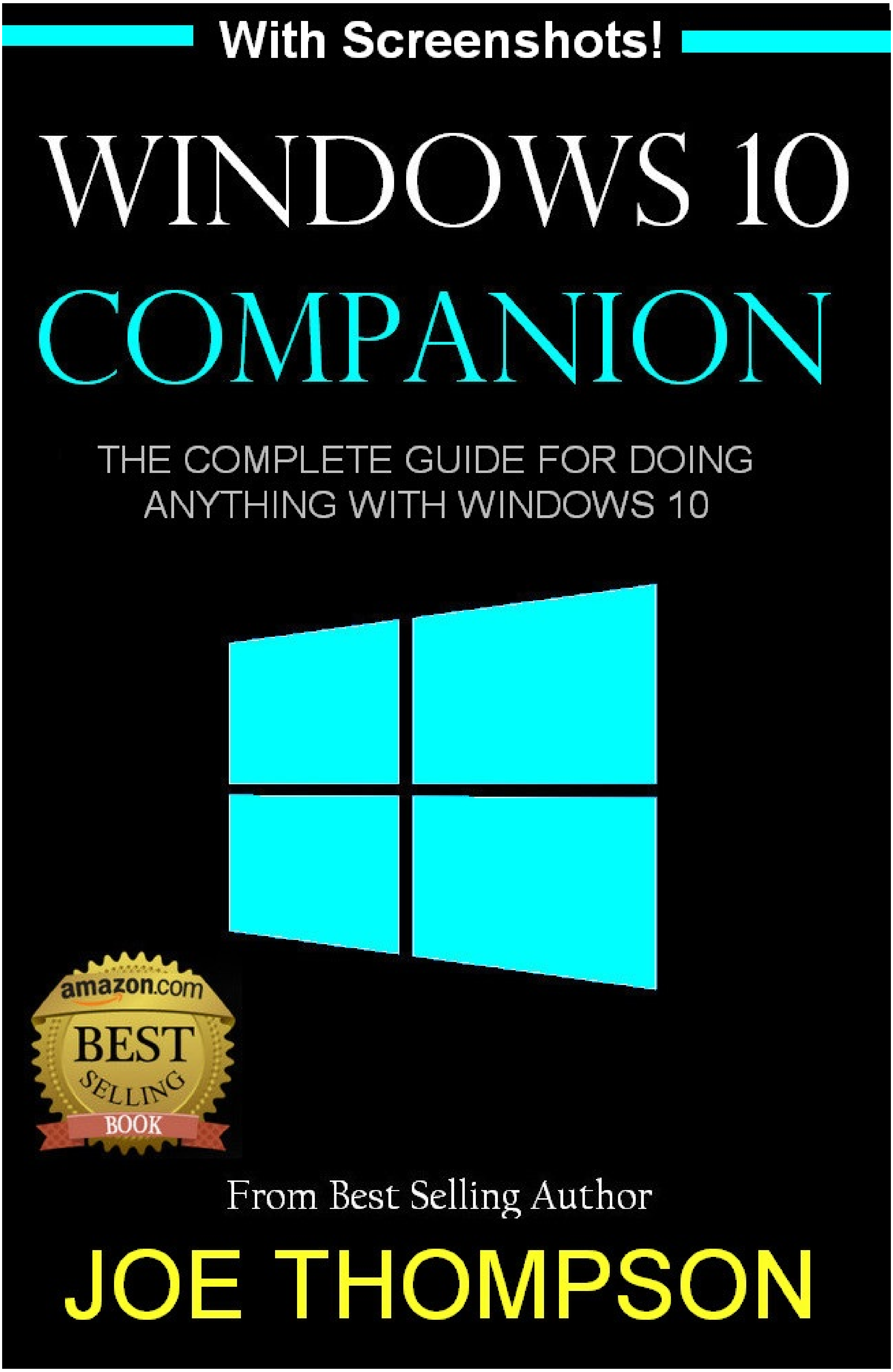
JOE THOMPSON

Python's Companion

The Most Complete Step-by-Step Guide to Python Programming



Joe Thompson’s Books

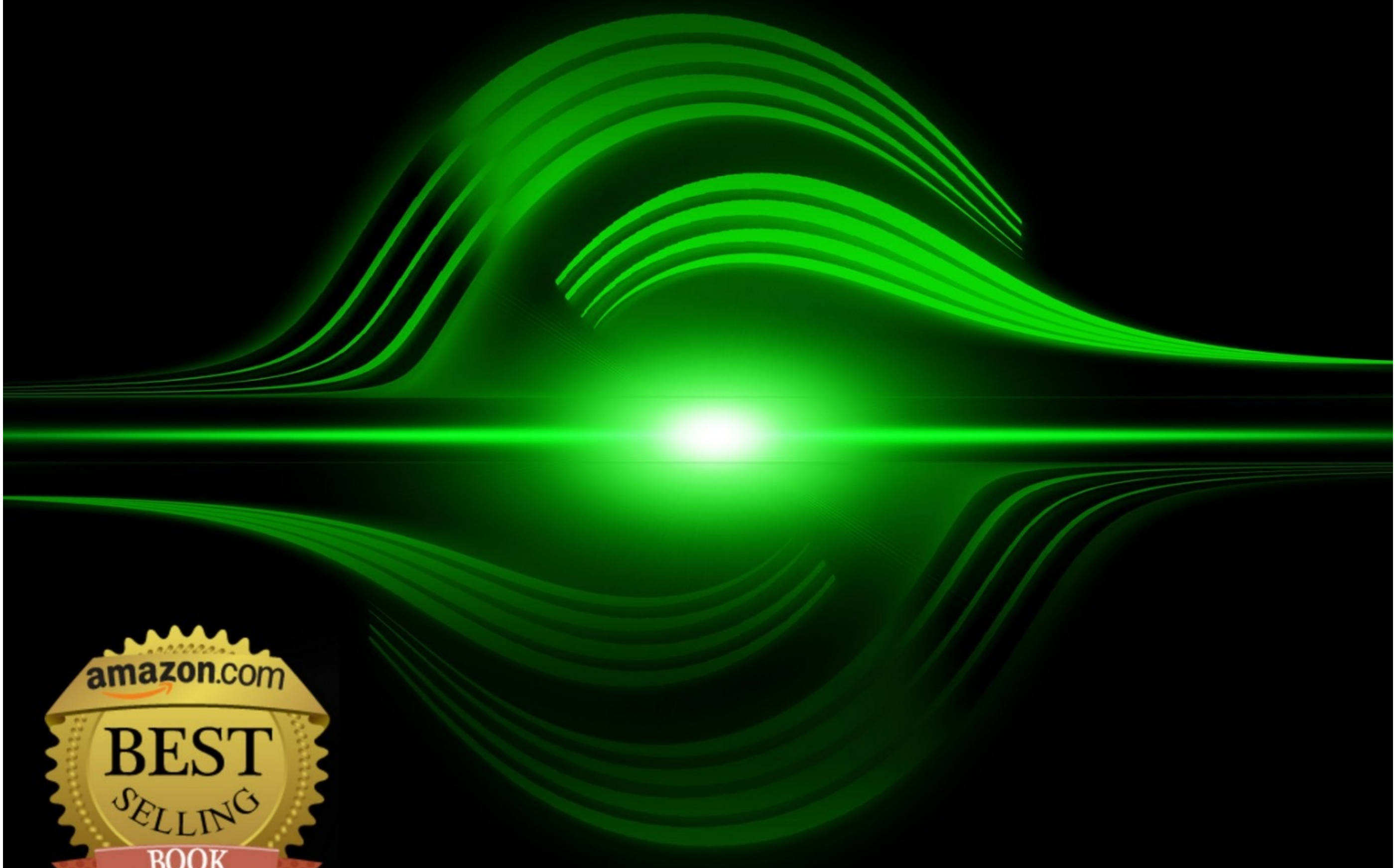


With Screenshots!
WINDOWS 10
COMPANION
THE COMPLETE GUIDE FOR DOING
ANYTHING WITH WINDOWS 10
amazon.com
BEST
SELLING
BOOK
From Best Selling Author
JOE THOMPSON

With Screenshots!

EXCEL 2016 COMPANION

THE COMPLETE GUIDE FOR DOING ANYTHING WITH EXCEL 2016



amazon.com BEST SELLING BOOK

From Best Selling Author

JOE THOMPSON

To learn Excel 2016 Step-by-Step <Click Here>

Table Of Contents

Introduction

An Overview of Python

Step 1: Installing Python

Installing Python in Windows

Which version should I use?

Installing Python in Mac

Running the Installation file

Starting Python

IDLE versus the command line interface (CLI)

IDLE

The Command Line Interface (CLI)

Different ways to access Python’s command line

If you’re using Windows

If you’re using GNU/Linux, UNIX, and Mac OS systems

Step 2: Working with IDLE

The Python Shell

The File Menu

The Edit menu

The Shell Menu

The Debug Menu

The Options Menu

The Window Menu

The Help Menu

Writing your First Python Program

Accessing Python’s File Editor

Typing your code

Saving the File

Running the Application

Exiting Python

Step 3: Python Files and Directories

The mkdir() Method

The chdir() Method

The getcwd() Method

The rmdir() Method

Step 4: Python Basic Syntax

Python Keywords (Python Reserve words)

Python’s Identifiers

Five rules for writing identifiers

A Class Identifier

Naming Global Variables

Naming Classes

Naming Instance Variables

Naming Modules and Packages

Naming Functions

Naming Arguments

Naming Constants

Using Quotation Marks

Statements

Multi-line statements

Indentation

Comments

Docstring

Step 5: Variables and Python Data Types

Variables

Memory Location

Multiple assignments in one statement

Assignment of a common value to several variables in a single statement

Data Types

Boolean Data Type

Step 6: Number Data Types

Integers (int)

Normal integers

Octal literal (base 8)

Hexadecimal literal (base 16)

Binary literal (base 2)

Converting Integers to their String Equivalent

integer to octal literal

integer to hexadecimal literal

integer to binary literal

Floating-Point Numbers (Floats)

Complex Numbers

Converting From One Numeric Type to Another

To convert a float to a plain integer

To convert an integer to a floating-point number

To convert an integer to a complex number

To convert a float to a complex number

To convert a numeric expression (x, y) to a complex number with a real number and imaginary number

Numbers and Arithmetic Operators

Addition (+)

Subtraction (-)

Multiplication (*)

Division (/)

Exponent (**)

Modulos (%)

Relational or Comparison Operators

Assignment Operators

= Operator

add and +=

subtract and -=

multiply and *=

divide and /=

modulos and %=

floor division and //=

Bill Calculator

Built-in Functions Commonly Used with Numbers

abs(x)

max()

min()

round()

Math Methods

Math.ceil(x)

Math.floor(x)

Math.fabs()

Math.pow()

Math.sqrt()

Math.log()

Step 7: Strings

Accessing Characters in a String

String Indexing

The Len() Function

Slicing Strings

Concatenating Strings

Repeating a String

Using the upper() and lower() functions on a string

Using the str() function

Python String Methods

The replace() method

Case Methods with String

Upper()

Lower()

Swapcase()

Title()

Count() method

The find() method

Isalpha()

Isalnum()

Isidentifier()

The join() method

Lstrip() method

Rstrip() method

Strip([chars])

Rfind() method

Index() method

Rindex() method

Zfill() method

Rjust() method

Ljust() method

Center() method

Endswith() method

Startswith() method

Iterating Through a String

Step 8: Output Formatting

The print() function

Using the str.format() method to format string objects

Other Formatting Options

‘<’

‘>’

‘^’

‘0’

‘=’

Step 9: Lists

Accessing Elements on a List

Indexing

Negative Indexing

Slcing Lists

Adding Elements to a List

Changing Elements of a List

Concatenating and Repeating Lists

Inserting Item(s)

Removing or Deleting Items from a List

Sorting Items on a List

Using the count() Method on Lists

Testing for Membership on a List

Using Built-in Functions with List

Len()

Max()

Min()

Sum()

Sorted()

List()

Enumerate()

List Comprehension

Step 10: Tuples

How to Create a Tuple

Accessing Tuple Elements

Indexing

Negative Indexing

Slicing a Tuple

Changing, Reassigning, and Deleting Tuples

Replacing a Tuple

Reassigning a Tuple

Deleting a Tuple

Tuple Membership Test

Python Tuple Methods

Count(x)

Index(x)

Built-in Functions with Tuples

Len()

Max()

Min()

Sorted()

Sum()

Tuple()

Enumerate()

Iterating through a Tuple

Tuples vs. Lists

Step 11: Sets

Creating a Set

Changing Elements on a Set

Removing Set Elements

Set Operations

Set Union

Set Intersection

Set Difference

Set Symmetric Difference

Set Membership Test

Using Built-in Functions with Set

Len()

Max()

Min()

Sorted()

Sum()

Enumerate()

Iterating Through Sets

Frozenset

Step 12: Dictionary

Accessing Elements on a Dictionary

Adding and Modifying Entries to a Dictionary

Removing or Deleting Elements from a Dictionary

The pop()method

The popitem() method

The clear() method

Other Python Dictionary Methods

Update(other)

Item() method

Values() method

Keys() method

Setdefault() method

Copy() method

The fromkeys() method

Dictionary Membership Test

Iterating Through a Dictionary

Using Built-in Functions with Dictionary

Lens()

Sorted()

Creating a Dictionary with the dict() function

Dictionary Comprehension

Step 13:Python Operators

Arithmetic Operators

Assignment Operators

Relational or Comparison Operators

Logical Operators

Identity Operators

Membership Operators

Bitwise Operators

Understanding the Base 2 Number System

Precedence of Operators

Step 14:Built-in Functions

The range() function

The input() Function

Password Verification Program

Using input() to add elements to a List

The print() Function

abs()

max()

min()

type()

Step 15: Conditional Statements

if statements

if…else statements

if…elif…else statements

nested if…elif…else statements

Step 16: Python Loops

The for Loop

For Loop with string:

For Loop with list

for loop with a tuple

Using for loop with the range() function

The While Loop

Break Statement

Continue Statement

Pass Statement

Looping Techniques

Infinite loops (while loop)

Loops with top condition (while loop)

Loops with middle condition

Loops with condition at the end

Step 17: User-Defined Functions

1. def keyword

2. function name

3. parameters

4. colon (:)

5. docstring

6. statement(s)

7. return statement

Calling a Function

Using functions to call another function

Program to Compute for Weighted Average

Anonymous Functions

Lambda functions with map()

Lambda functions with filter()

Recursive Functions

Scope and Lifetime of a Variable

Step 18: Python Modules

Importing a Module

Python's Math Module

Displaying the Contents of a Module

Getting more information about a module and its function

The Random Module

Usage of Random Module

Random Functions

Universal Imports

Importing Several Modules at Once

Step 19: Date and Time

Formatted Time

Getting Monthly Calendar

The Time Module

The Calendar Module

calendar.firstweekday( )

calendar.isleap(year)

calendar.leapdays(y1, y2)

calendar.month(year, month, w=2, l=1)

calendar.monthcalendar(year, month)

calendar.monthrange(year, month)

calendar.prmonth(year, month, w=2, l=1)

calendar.setfirstweekday(weekday)

calendar.weekday(year, month, day)

Datetime

Step 20: Namespaces

Scope

Step 21: Classes and Object-Oriented Programming

Defining a Class

Creating an Object

The __init__() method

Instance Variables

Adding an attribute

Deleting Objects and Attributes

Modifying Variables within the Class

Inheritance

Multiple Inheritance

Multilevel Inheritance

Step 22: Python Iterators

Creating a Python Iterator

Step 23: Python Generators

Step 24: Files

The File Object Attributes

File Operations

The Open() function

Writing to a File

Closing a File

Opening, Writing to, and Closing a Text File

Reading a Python File

The readlines() method

Line by Line Reading of Text Files with the ‘while’ loop

Line by Line Reading of Text Files using an Iterator

The ‘with statement’

Appending Data to a File

Renaming a File

The rename() method

Deleting a File

Binary Files

File Methods

File.writer(str)

File.writelines(sequence)

File.readline(size)

File.readlines()

File Positions: file.tell() and file.seek

File.tell()

File.seek()

File.read(n)

File.truncate([size])

File.flush()

File.close()

File.isatty()

File.fileno()

Step 25: Handling Errors or Exceptions

Syntax Errors

Runtime Errors

Built-in Exceptions

Catching Exceptions

try and except

try…finally

Python Cheat Sheets

Variable Assignment

Accessing Variable Values

Python Operators

Arithmetic Operators

Assignment Operators

Relational Operators

Logical Operators

Identity Operators

Membership Operators

Bitwise Operators

Strings

Lists

Tuple

Dictionary

Sets

Loops

Conditional Statements

if…else

if…elif…else

Built-in Functions

User-Defined Functions

Classes

Files

Text File Opening Modes

Binary File Opening Mode

File Operations

Date and Time

Python Modules

Help!

Google

FAQ

IRC (Internet Relay Chat)

Other Best Selling Books You Might Like!

Conclusion

-

Copyright

Introduction

I want to thank you and congratulate you for downloading the book, *“Python’s Companion”*.

This book contains proven steps and strategies to help beginners learn Python Programming quickly and easily. It is designed to be a practical, step-by-step tutorial of essential Python programming concepts for self-learners from beginner to intermediate level.

It uses a straightforward approach that focuses on imparting the important ideas without the heavy programming jargon. Python, after all, is a language with simple and easy-to-learn syntax.

The book features various Python programs as examples as well as a concise explanation of the different aspects of Python Programming. By the time you finish the book, you will be equipped with the necessary skills to create useful and practical codes on your own.

IMPORTANT

Although we have made this book as easy as possible to use, you might find yourself stuck at some point. When programming, you might, for example, get the “Error” message and not know what to do. The last Chapter focuses on helping you solve these issues. Do not hesitate to use it and **NEVER** give up!

If you happened to like my book please leave me a quick review on Amazon. I personally read every single comment on the site. Your feedback is very important to me as it will help improve the book and ensure your satisfaction.

Thanks again for downloading this book. I hope you enjoy it!

An Overview of Python

Python was developed by Guido van Rossum at the National Research Institute in the Netherlands in the late 80s. Its first version was officially released in 1991. He derived its core syntax from the ABC language, borrowed heavily from C, and borrowed partly from other languages such as C++, Modula-3, Unix shell, SmallTalk, and Algol-68.

Python is a high-level, object-oriented, interactive, interpreted, and general-purpose programming language.

A high level scripting language, Python codes are designed to be highly readable. Its programs are written in nearly regular English and are neatly indented. It uses English keywords, concise codes, and simple syntax.

It is a multi-paradigm programming language that supports structured, functional, aspect-oriented, and object-oriented programming methods.

Python is an object-oriented programming language. It emphasizes the use of objects over functions in contrast with languages that are procedure-oriented. It supports user-defined classes, inheritance, multiple inheritance, and methods binding at runtime.

Python is interactive. You can write your program directly on the Python prompt and generate immediate feedback. You can use the prompt to test bits of codes.

Python is both a compiled and interpreted language. Python is an interpreted language because it has to convert codes into machine-readable byte codes before it can run them. It is also a compiled language because behind the scenes, Python is implemented with a combination of an interpreter and bytecode compiler. It performs the compilation implicitly as it loads the modules. In addition, some standard processes require the availability of the compiler at runtime.

Python is a powerful and flexible language. You can use it to automate and simplify complicated tasks at work or at home. You can develop large web applications, GUIs, and interesting games with it.

One of the main reasons for Python's popularity is its extensive standard library which contains more than a hundred modules. Users can access these modules easily with a simple import statement. The most commonly used ones include network programming, internet protocols such as HTTP, SMTP, and FTP, mathematical functions, random functions, regular expression matching, operating systems, threads, networking programming, GUI toolkit, HTML parsing, XML processing, and email handling.

Although it borrows heavily from C, Python differs in two major areas with C: the use of indentation to structure blocks of codes and the use of dynamic typing.

In C, statements are grouped with the use of curly braces and a trailing semicolon. In Python, statements are organized into a block with the use of indentation.

In C, variables are declared and assigned a specific type before they can be used in a program. In Python, variables are just names that point to objects stored in memory. They need not be declared before they are assigned. They can be reassigned and they can

change types anywhere in the program.

Other features that contribute to its popularity are its support for garbage collection and easy integration with other languages such as C, C++, Java, CORBA, ActiveX, and COM.

Because of its readability and the use of plain English in most of its construction, Python is an easy-to-learn language for many people. While some background in programming is desirable, you don’t need it to learn Python. The language is so easy to learn that many experts recommend Python to students and beginners who are learning to program for the first time. Python is, in fact, for those who are new to programming who want to make powerful and useful programs within a short span of time.

This book is organized as a step-by-step material to learning Python. Each step is designed to build the required skills to progress to the more advanced topics in its study.

Step 1: Installing Python

Python is an open-source programming language which can be freely downloaded in the website of Python Software Foundation.

Installing Python in Windows

To install Python in Windows, you must first download the installation package of your preferred version. This is the link to the different installation packages for Windows.

https://www.python.org/downloads/

On the landing page, you will be asked to select the version you want to download. There are **TWO HIGHLIGHTED VERSIONS**:

1. the latest version of Python 3, Python 3.5.1
2. the latest version of Python 2, Python 2.7.11.



If you're looking for an older release, scroll further down the page to find links to other versions.

Which version should I use?

Python 2 is the older version and was published in late 2000. Python 3 was released in late 2008 to address and amend intrinsic design flaws of previous versions of the language. It is regarded as the future of Python and is the version of the language that is currently in development. If you want to read more about it's differences and which one you should use you can read about it on <Wiki.Python>.

PLEASE NOTE

This book uses version 3.5.2 of Python.

Installing Python in Mac

If you're using a Mac, you can download the installation package from this link:

https://www.python.org/downloads/mac-osx/

Running the Installation file

After completing the download step, you can install Python by clicking on the downloaded .exe file. Setup is automatic. Standard installation includes IDLE, pip, and documentation.

Starting Python

IDLE is the integrated environment that you can use to write and run your programs in. Both IDLE and command line interface is installed when you install Python.

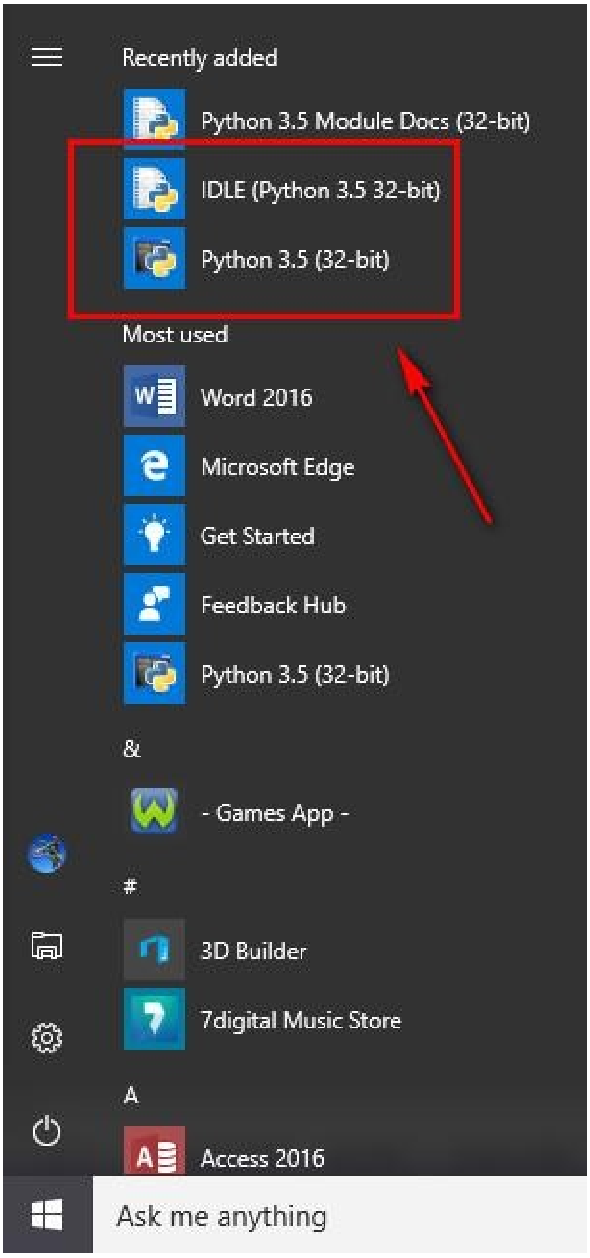


Figure 1. Once you install Python into your computer both **IDLE** and **command line interface** are installed.

IDLE versus the command line interface (CLI)

IDLE

IDLE offers a more advanced interface with many options for working with Python. You can use it in the same way that you would use a command line and more. As an integrated environment, it offers its own text editor which is easily accessible by clicking on a menu item. Moreover, it has editing options which make writing programs or simply typing codes more efficient.

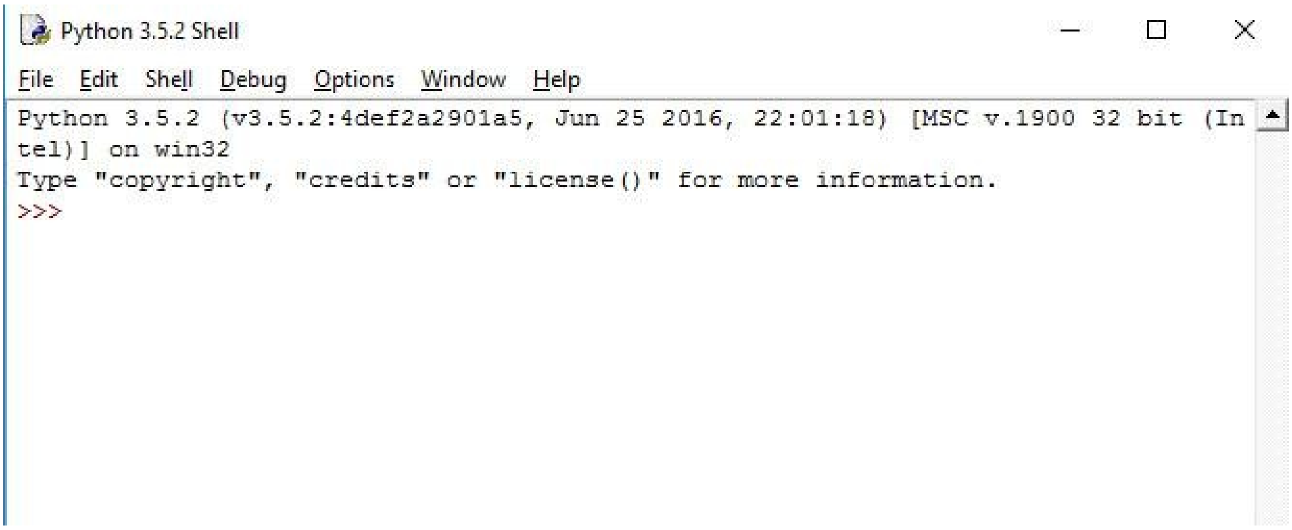


Figure 2. IDLE' s interface.

The Command Line Interface (CLI)

The **command line** is useful for trying out lines of codes.

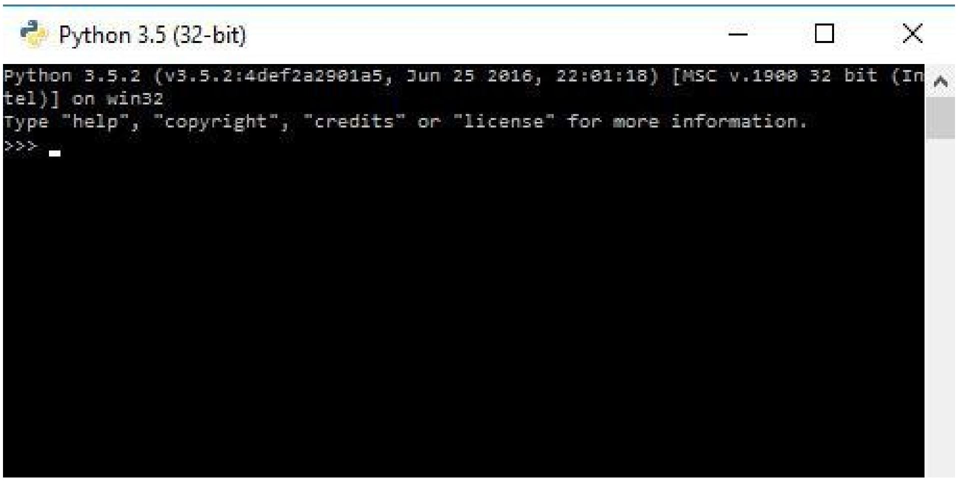


Figure 3. CLI's interface.

Different ways to access Python's command line

There are different ways to access Python's **command line** or **IDLE** depending on the operating system installed on your machine:

If you're using Windows

You can start the Python command line by clicking on its icon or menu item on the Start menu.

You may also go to the folder containing the shortcut or the installed files and click on the Python command line.

If you’re using GNU/Linux, UNIX, and Mac OS systems

You have to run the Terminal Tool and enter the Python command to start your session.

Step 2: Working with IDLE

IDLE, an integrated development environment (IDE), was created for use with Python. It is bundled with and installed alongside the Python interpreter. IDLE offers a simple graphical user interface with helpful features that make program development more convenient and intuitive.

IDLE is an efficient, powerful, and flexible platform for exploring, testing, writing, and implementing your programs in Python.

In interactive mode, Python evaluates every expression entered as it simultaneously runs statements stored in active memory. It then provides instant feedback. The interactive mode allows you test run lines or pieces of codes before you actually use them in your program. Use it as your playground in your study of Python's syntax.

In the script mode, the interpreter runs scripts or codes saved with a **.py** file extension. The script mode is also called the **normal mode**.

Following are the most important features of IDLE:

- Python Shell
- multiple-window text editor
- auto-completion
- smart indent
- syntax highlighting
- integrated debugger

The Python Shell

The **Python Shell** offers users an efficient and user-friendly interface with an easy-to-navigate drop down menu and useful editing features. The Python Shell provides so much more functionality than the command line interface. You can use it to run your programs as well as work interactively with Python. Simply type an expression or statement on the >>> prompt and press **enter**. You can easily scroll back, review previous command entered, and copy paste previous statement to the current prompt to run it again or make necessary modifications before running the code again.

Python Shell main menu shows the following options:

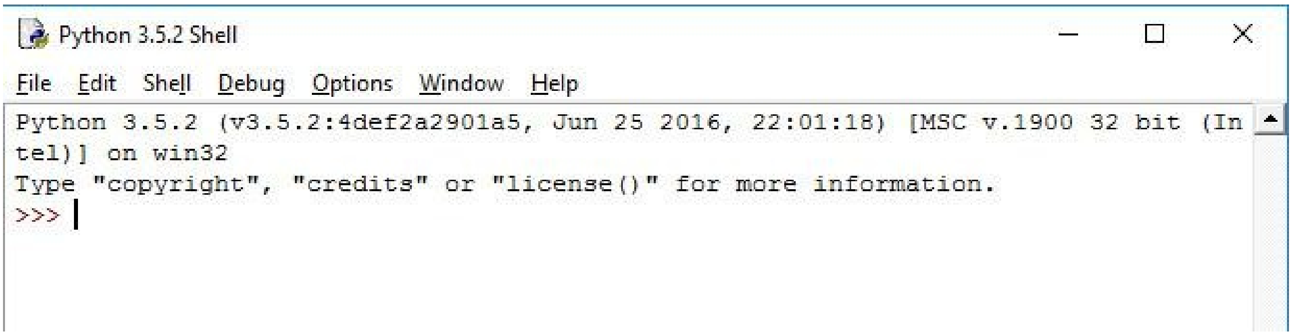


Figure 4. Python Shell's main menu includes: File, Edit, Shell, Debug, Options, Window and Help.

Just to get a feel of the Python Shell and the IDLE environment, use the **print()** function to instruct the interpreter to output "Hello, World!" on your monitor.

After the >>> prompt, type the expression:

print("Hello, World!")

PLEASE NOTE

A prompt is a text or a symbol used to represent the system's readiness to perform the next command. The symbols ">>>>" means the computer is waiting for a new input.

As you press ENTER, Python reads, interprets, and outputs Hello, World! onscreen:

```
>>> print("Hello, World!")
Hello, World!
>>> print
>>>
```

Now, do away with the print() statement and just type "Hello, World!" on the >>> prompt. Watch how Python returns the string:

```
>>>"Hello, World!"
'Hello, World!'
```

```
>>>
```

The Python Shell isn't just a place for trying out your codes. It is a place for solving mathematical expressions as well. Python supports basic arithmetic operators and you can invoke them right at the >>> prompt. Just type the expression and press ENTER. Try it out:

```
>>>12 * 12          #12 times 12
144

>>> 82/4          #82 divided by 4
20.5

>>>3**3          #3 raised to the 3rd power
27
>>>
```

PLEASE NOTE

The sentences on the right with the "#"(#12 times 12, #82 divided by 4 and #3 raised to the 3rd power), are not part of the expression. They are ways for me to better explain Python's expressions.

The File Menu

The File menu provides basic options for creating a new file, opening and saving a file or module, browsing class and path, accessing recent files, printing the window, and exiting Python Shell.

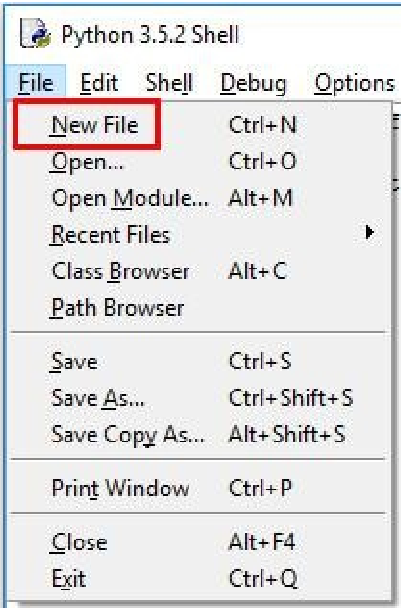


Figure 5. Here are the options available once you press on the File Menu.

Clicking on the "**New File**" option (see figure 5) opens a text editor that you can use to type, modify, save, and run your program. Your program's output is displayed on the Python Shell. Python's built-in text editor is a basic one with many useful features and a menu that almost mimics that of the Python Shell. It offers the following menu options:

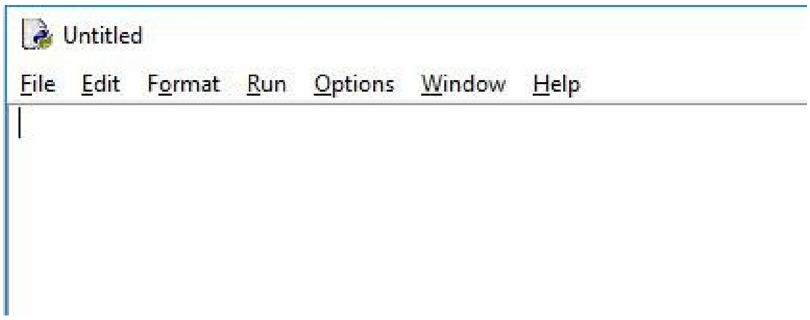


Figure 6. Once you select "New File", the menu option offers; File, Edit, Format, Options, Window and Help.

Take note that the above menu options lack the Shell and Debug options found in Python Shell. It has, however, the Format and Run options which are not found in the Python Shell.

The Format menu offers useful options for indenting/dedenting, toggling/untoggling comments, setting tabs and indent width, paragraph formatting, and stripping trailing whitespace.

The Run menu offers options for checking and running your module. The output is displayed on the Python Shell.

The other options, namely File, Edit, Options, Windows, and Help, offer practically the same features that you can find on Python Shell.

The Edit menu

Python Shell's Edit menu provides practical options for copying, cutting, pasting, undoing, redoing, selecting, searching for and/or replacing a word on a file.

The Shell Menu

The Shell menu provides options for restarting the Shell and viewing the most recent Shell restart.

The Debug Menu

The Debug menu opens the Debugger and the Stack Viewer. It can be used to trace the line when Python raises an error. Clicking on the Debugger opens an interactive window that will allow users to step through a running program.

The Options Menu

The Options menu opens various preferences for configuring the font, tabs/indentation width, keys, highlighting, and other general options in Python Shell.

The Window Menu

The Window menu offers options for adjusting zoom height and for moving between Python Shell and open text editor windows.

The Help Menu

The Help menu provides access to Python's documentation and help files.

Writing your First Python Program



Now that you're more or less familiar with Python's working environment, it's time to create your **first program with IDLE**. You can use another text editor, but for simplicity, you can use **Python Shell'**s built-in file editor. This is where you'll be typing your first program. When you're done, you can save the file in a specific directory on your hard disk or in another storage device.

Accessing Python's File Editor

To open a new file editor window on the Python Shell:

Click on File ->Choose New File

Python's file editor is a typical text editor with basic editing and formatting options.

Typing your code

You can start working on your first program by typing statements or commands on the file editor.

For this purpose, you can type the universal program **"Hello, World!"**

Type the following on the text editor:

```
>>>print("Hello, World!")
>>>
```

Saving the File

Saving the file prepares it for further processing by Python. It also allows you to work on your application on a piece meal basis. You can store the file for future use or editing once

it’s saved on your storage device. To save the file:

Click on File ->Choose “Save as”

The default installation folder for saving a program is Python’s installation folder. You can change this to your preferred file destination folder. Saving your file is important because you can’t possibly run your code without saving it. Python programs automatically get a **.py** extension in IDLE. For instance, if you save your file in Python Shell’s text editor, you can name your program as: MyFirstProgram and Python will save it as **MyFirstProgram.py**.

Running the Application

The most basic level of a program’s success is for it to be run without error or exceptions by Python. When you run a program, you are instructing Python to interpret and execute your saved code.

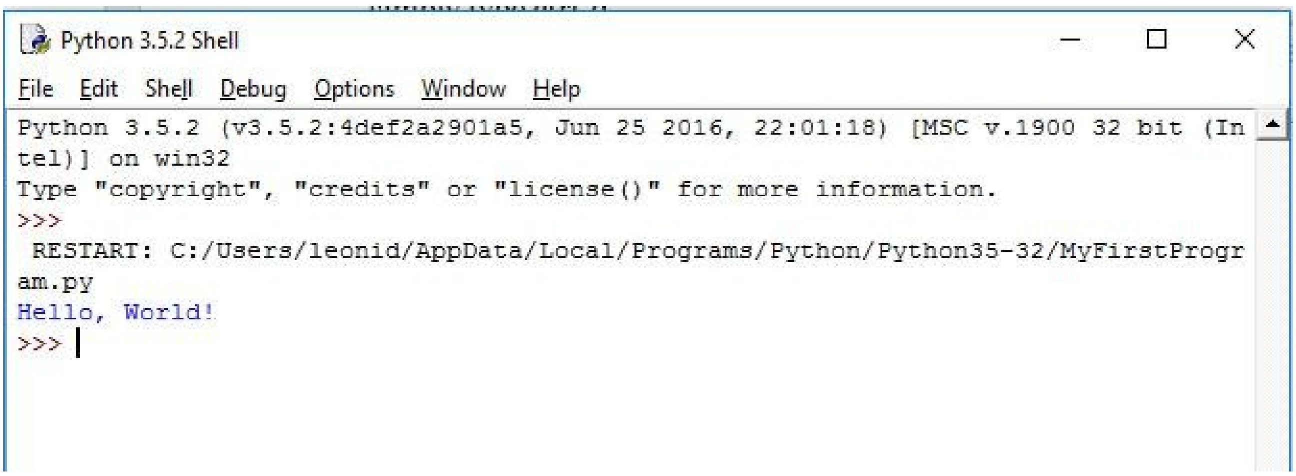
To run your program, click on the Run menu in the text editor. You’ll have these options:

Python Shell

Check Module

Run Module

To run your application, choose Run Module. If there are no errors, the Python Shell will display something like this:



If there are errors, Python will display the error message on the Python Shell.

Exiting Python

To exit from IDLE’s Python Shell, you can type exit() or quit() and press enter or simply type ctrl-d.

If you’ve chosen to access the command line in the Power Shell instead of IDLE, you can type either exit() or quit() and press enter.

Step 3: Python Files and Directories

Programs and other data are saved in files for future access and use. Files are stored in different directories and should be accessed accordingly. Python handles files and directories with its **os module**. The **os module** contains many useful methods that will help users create, change, or remove directories.

The mkdir() Method

The mkdir() method is used to create new directories in the present directory. It requires one argument: the name of the directory you want to create.

The **syntax** is:

```
os.mkdir(“newdir”)
```

For example, if you want to create a new directory “trial”, you can use the following code:

```
>>>import os
```

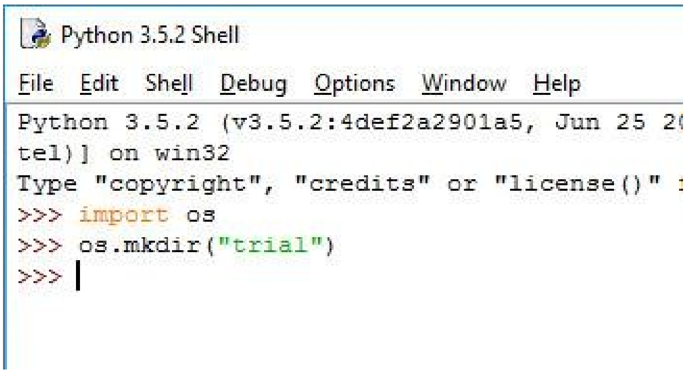
PLEASE NOTE

Don’t worry if you do not understand this part yet. If you are completely new to programming you can skip this part for now and move to Step 5. The code “import” is used to import a module. We will later discuss how to import a module in Step 18.

Now create a directory “trial”:

>>>os.mkdir("trial")

>>>



The chdir() Method

You will want to change the current directory if you need to work on a file that is stored in another directory. The chdir() Method can be used to change the present directory. This method takes only one argument and has the following **syntax**:

os.chdir("newdir")

For example, if you're current directory is Python/sciapp and you want to change to another directory, Python/mathapp, here is how you will use os.chdir():

>>>import os

Change a directory to "/Python/mathapp":

>>>os.chdir("/Python/mathapp")

>>>

The getcwd() Method

The current working directory is the directory that Python constantly holds in memory. It is an ever present property. When you issue commands like 'import file.txt', Python will search for the file in this directory. To display the current working directory, you can use the os module's getcwd() method with the **syntax**:

os.getcwd()

Example:

```
>>>import os
```

This gives the location of current working directory:

```
>>>os.getcwd()
>>>
```

The rmdir() Method

The rmdir() method can be used to delete the directory. It takes one argument: the name of the directory to be deleted.

Before you can remove a directory, all contents should first be removed.

The **syntax** for rmdir() method is:

os.rmdir(‘dirname’)

Take note that the full name of the directory should be provided. If not, the interpreter will search for the directory in the current working directory.

Here’s an example of its usage:

```
>>>import os
```

Now remove the “/pyprog/mathapp” directory:

```
>>>os.rmdir( “/pyprog/mathapp”  )
```

Step4: Python Basic Syntax

In order for the Python interpreter to process and execute your instructions properly, you must write your code in the format that it can understand. In programming, **syntax** refers to the set of rules that define the correct form, combination, and sequence of words and symbols.

PLEASE NOTE

An interpreter is a program that reads and executes codes. Python is a great example of an interpreter.

Python Keywords (Python Reserve words)

Keywords are words that Python reserves for defining the **syntax** and **structure** of the Python language. Hence, you **can't use** them as a regular identifier when naming variables, functions, objects, classes, and similar items or processes. Take note of **these 33 keywords** and avoid using them as identifier to avoid errors or exceptions:

and	or	not
assert	yield	except
break	continue	pass
class	def	del
finally	global	nonlocal
if	elif	else
import	lambda	from

in	is	try
True	False	None
while	for	return
with	raise	as

PLEASE NOTE

- All the keywords except **True**, **False** and **None** are in lowercase and they must be written as it is.
- Keyword are case sensitive.

Python's Identifiers

An **identifier** is a name given to a **variable**, **class**, **function**, **string**, **list**, **dictionary**, **module**, and other **objects.** In python, it helps differentiating one entity from another.

PLEASE NOTE

These entities (variable, class, function, string, list, dictionary, modules and objects) will be later explained in this book.

Variables: Step 5

Class: Step 21

Five rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Here are some examples:

- C
- c
- my_variable,
- my_var9
- UPPERCASE
- lowercase
- UPPERCASE_WITH_UNDERSCORES
- lowercase_with_underscores

- CamelCase or CapWords
- mixedCase

2. An identifier cannot start with a digit.

   For example:
   1lowercase is invalide but **lowercase1** is accepted.

3. Keywords cannot be used as identifiers.

4. Identifier may not contain special symbols like !, @, #, $, % etc

5. Identifier can be of any length.

Avoid using the letters “O” (uppercase), “l” (lowercase letter l), or “I” (uppercase letter I) as a single letter identifier as they may cause unnecessary confusion with the numbers 0 (zero) and 1 (one) in some fonts.

An identifier may consist of several words but for better readability, use a single underscore to separate each unit.

Examples:

my_secret_variable

my_great_function

A Class Identifier

A class identifier may be a CamelCase name, that is, a name consisting of two or more words which all start in capital letters and are joined together without underscores or spaces.

Examples:

MyFamousCollection

TheFabulousList

Abbreviations in CamelCase names should be capitalized. For example: HTTPServerError.

You cannot use keywords as identifier. However, you may use a trailing underscore to distinguish the name from a keyword.

Example:

class_

Naming Global Variables

A global variable name should be written in lowercase. A global variable name consisting of two or more words should be separated by an underscore.

The following naming conventions are specific to each object in Python. Read them and use them as reference as you go through each lesson.

Naming Classes

Apply the UpperCaseCamelCase convention when naming a class. In general, built-in classes are usually written in lowercase. Exception classes normally end in “Error”.

Naming Instance Variables

An underscore should separate instance variables consisting of several words. Variable names should be written in lower case. A non-public instance variable identifier starts with one underscore. An instance name that requires mangling starts with two underscores.

Naming Modules and Packages

Names for modules should be all in lower case. Preferably, module names should be short one-word names. Although it is discouraged, multiple words may be used if required for better readability. They should be separated by a single underscore.

Naming Functions

A function name should consist of letters in lowercase. When using multiple words as function name, they should be separated by underscores to enhance readability.

Naming Arguments

You should always use the word “self” as the first argument in an instance method and “cls” as the first argument in a class method.

It is generally preferable to use a single trailing underscore over an abbreviation or name corruption when you want to distinguish an argument’s name from a reserved keyword.

Naming Constants

Constants are written in all uppercase letters with underscores to separate multiple-word names.

Examples:

TOTAL

MAX_OVERFLOW

Using Quotation Marks

Quotation marks are used to indicate string literals. You can use single (′), double (“), or triple (”’) quotes but you must observe consistency by using the same quotation marks to begin and end the string.

Examples:

‘Joshua’

“occupation”

”‘age:”’

‘These are your options’

Statements

Statements are expressions that you write within a program which are, in turn, read and

executed by the Python interpreter. Python supports statements such as if statement, for statement, while statement, break statement, and assignment statements.

Multi-line statements

Statements may sometimes be too long to fit in a single line and are thus written across several lines. To implicitly tell Python that a multiple line expression is a single statement, you can enclose it in braces{}, brackets[], or parentheses ().

Example #1:

```
my_alphabet_letters = (“a”, “b”, “c”, “d”, “e”,
                        “f”, “g”, “h”,“i”, “j”,
                “k”, “l”,  “m”, “n”, “o”, “p”)
```

Example #2:

```
my_colors = [“blue”, “red”, “yellow”,
                        “green”, “orange”, “violet”,
                        “pink”, “black”, “white”]
```

The above structure illustrates the implicit way to let Python know that it should expect a multiple-line statement. To explicitly indicate continuation, you can use a backslash (\) at the end of each line:

```
my_alphabet_letters = (“a”, “b”, “c”, “d”, “e”, \
                            “f”, “g”, “h”, “i”, “j”,  \
                                    “k”, “l”,  “m”, “n”, “o”, “p”
```

Indentation

Python programs are structured through white spaces or indentation. Hence, if you have been using Java, C, or C++ for a while, you’ll never see the familiar curly braces {} around block codes. White spaces make Python programs look more organized and readable. In Python, a block starts from the same distance to the right and ends on the first unindented line. Starting a nested block is as easy as indenting further to the right.

By convention, programmers use four white spaces instead of tabs to indent a block of

code. While this level of indentation is not strictly imposed in Python, consistency of indentation within the block code is required if you want your program to run as expected.

To illustrate how indentation is implemented, take a look at this program segment:

```
def tool_rental_fees (days):
    fees = 10 * days
    if days >= 7:
        fees -= 10
    elif days >= 3:
        fees -= 5
    return fees
```

Comments

Comments are notes written within programs to describe a step, process, or to just basically include important information. Comments provide documentation to your work and are useful during review or program revisit. It will make work a lot easier for you and other programmers to evaluate the program months or years down the line. A comment is introduced by a hash (#) symbol which tells the interpreter to ignore the line and proceed to the next statement.

```
#print out a welcome greeting
>>>print('Welcome to Happy Coders Club!')
>>>
```

For long comments that span over several lines, you can wrap the connected lines together with a hash (#) symbol at the beginning of each line:

```
#This long comment is necessary
#and it extends
#to three lines
```

Alternatively, you can use triple quotes at the start and end of a multiple-line comment.

```
“““This is an alternative
way of writing
multi-line comments”””
```

Docstring

A documentation string or docstring is used for documenting what a class or function does. You will find a docstring summary at the top of a block of code that defines a class, method, function, or module. Docstrings are typically phrases that start with a capital letter and end with a period. They may span over several lines and are enclosed within triple double quotes (”””). A docstring starts with a one-line summary which is written like an imperative statement.

Examples:

```
def triple_value(num):
    “““Function to get thrice the value of a number.”””
    return 3*num



class Documentation(object):

    “““Explain class docstring.

A class uses a distinct whitespace convention in multi-line docstrings where a
blank line appears before the opening quote and after the closing statement.
The closing quote is placed right after the blank line.

”””
```

Step 5: Variables and Python Data Types

Variables

A **variable** is a reserved memory location that can be used to store values. This means that when you create a variable you reserve some space in the memory.

Whenever you create a variable, you are allocating space in a computer's memory, a storage that can hold values. Variables are given distinct names to identify their memory locations. These names are then used as a reference to instruct the computer to access, edit, save, or retrieve stored data.

Variable creation and management is a lot more flexible and straightforward in Python than in most other languages. You can easily create or declare a variable by using a syntactically appropriate name and assigning a value to it using the assignment operator (=). You need not even inform Python that you want a variable to contain a specific type of data. Python automatically identifies the data type based on the values you assign to the variable.

Study these examples of variable assignment statements:

employee =	"Jonathan Brown"
average_score =	"80.5"
counter =	"18"
comment =	"That input is incorrect."

The left operands (in grey) are the **names of the variables** while the right operands (in blue) refer to the **value assigned to the variable**. The use of the assignment operator tells Python that a variable is assigned to store a particular value. Hence, in the assignment statement "average score = 80.50", you are basically telling Python that the variable 'average score' is set to '80.50'. Python, in this case, knows that you are using the variable 'average score' to store a floating point number and there's absolutely no need to declare that this variable is going to hold a float.

Memory Location

To check the **memory location** of the above variables, you'll use the **id()** operator.

But first, open you IDLE and type the variables and their values showed above.

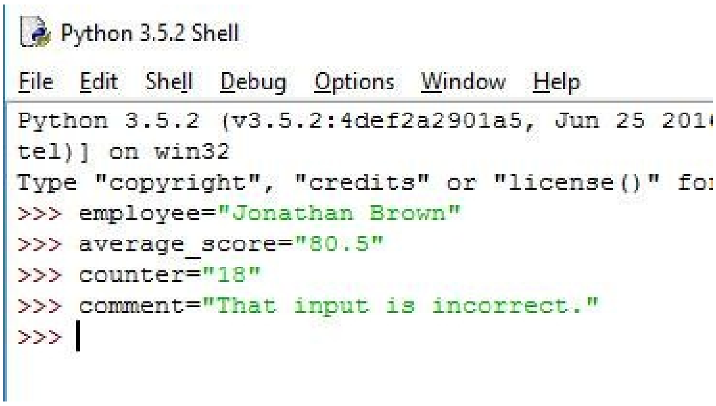


Figure 7. Here are the variable assignments statements.

Now you can type these assignments:

```
>>>id(employee)

>>>id(score)

>>>id(counter)

>>>id(comment)
```

Once you type your assigments and press ENTER, Python will give you the memory location:

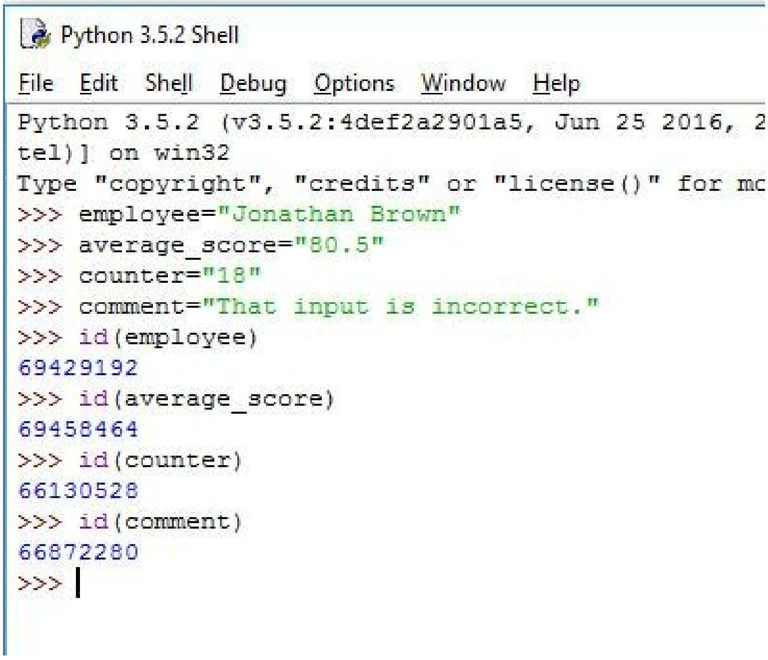


Figure 8. Below each assignment, in blue, you can see the memory location.

Multiple assignments in one statement

Python accepts multiple assignments in one statement with this **syntax**:

```
x, y, z = a, b, c
```

Multiple variable assignments follow positional order. To illustrate, assign a **string**, a **float**, and an **integer** to the variables ‘name’, ‘rating’, and ‘rank’:

IMPORTANT DEFINITIONS

String: A string is a list of characters in order. It can be a letter, a number, a space, or a backslash. There are no limits to the number of characters you can have in a string . You can even have a string that has 0 characters, which is usually called “the empty string.”. To tell Python you are using a string you must use single quotes (‘), double quotes (“) and triple quotes(“””).

Example: employee= “Jonathan Brown”. The string here is Jonathan Brown.

Integer : These are positive or negative whole numbers (they don’t include any decimal points). This will be discussed later in Step 6.

Example: 3 is an integer but 2.5 is not an integer.

Float : These are real numbers and are written with a decimal point dividing the integer and fractional parts. This will be discussed later in Step 6.

Example: 63.9 is a float

```
>>>name,rank,rating="Johnston",89.5,26
>>>
```

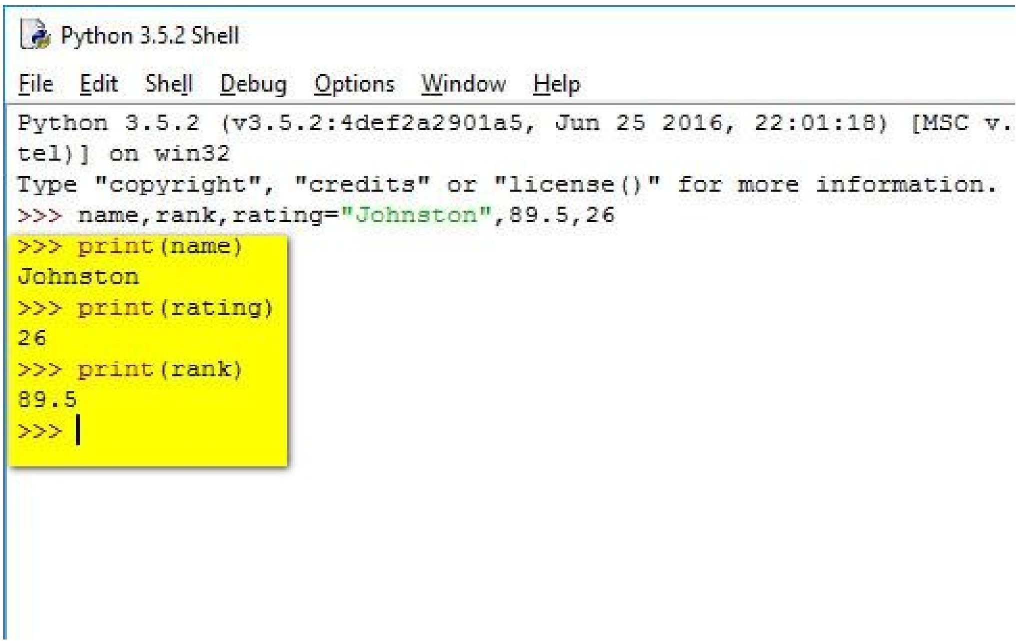
Now, use the **print** function to see the values stored in the above variables:

```
>>>print(name)
```

```
>>>print(rating)
```

```
>>>print(rank)
```

Here is what you will get:



In the above example, the variable ‘name’ holds the string “Johnston”, while the variables

‘rating’ and ‘rank’ hold the integer 26 and the float 89.5 respectively.

Assignment of a common value to several variables in a single statement

Python allows the assignment of a common value to several variables in a single statement with the **syntax**:

x = y = z = “apple”

The statement assigns the string “apple” to variables x, y, and z at the same time.

To check if the variables x, y, and z actually pertain to one and the same memory location, use the **id()** operator:

>>>id(x)

>>> id(y)

>>>id(z)

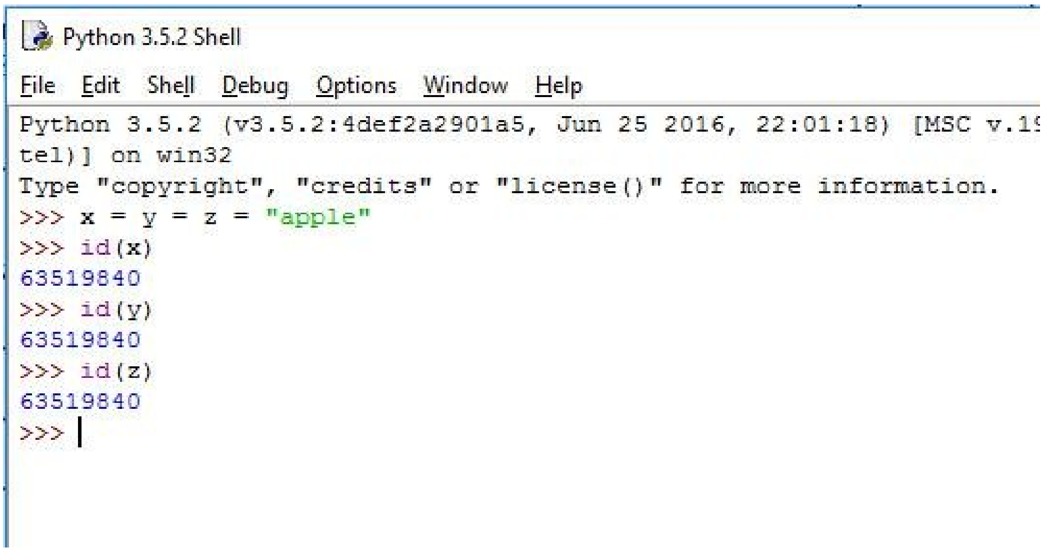


Figure 9. In this example you can see that the 3 variables pertain to the same memory location 63519840.

Just as easily as you have created and assigned values to a variable, you can change the value and corresponding data type stored in it in a simple reassignment process.

To illustrate, create a variable named ‘counter’ and assign the value of 50, an integer.

>>>counter = 50

To increase the value stored in the variable by 30, enter this statement on the Python prompt:

>>>counter += 30

To see how Python interprets your statement, use the print function to see the value stored in the variable ‘counter’:

>>>print(counter)

You should see this value on the succeeding line:

80

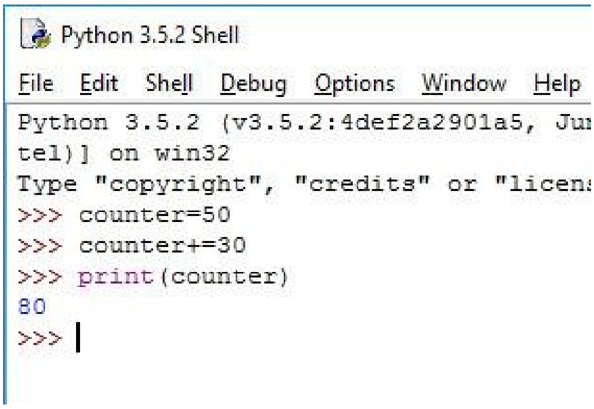


Figure 10. Once you have typed your statements, you should get 80 as a result.

The above example showed that you can easily replace the value stored in a variable with a single statement. The following example will demonstrate that you can also reassign a variable to store a value with a different data type with a simple reassignment statement. Assuming you now want the variable ‘counter’ to hold the string “upper limit”, just type the regular assignment statement you have learned earlier:

>>>counter = “upper limit”

Now, use the print function to see how it has affected the value stored in the variable ‘counter’:

>>>print(counter)

The output should show:

upper limit

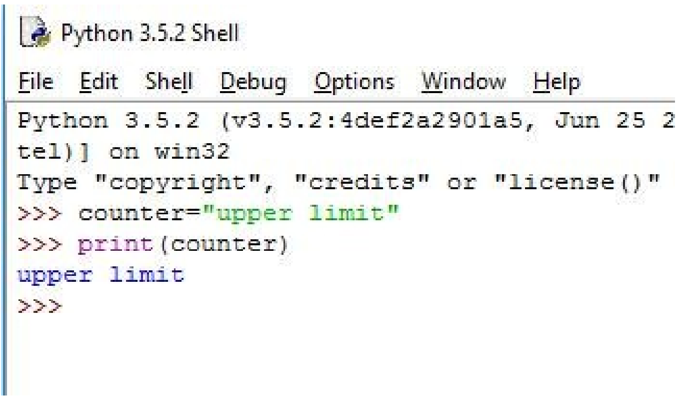


Figure 11. As you can see the output is upper limit.

Data Types

Programming involves a lot of data processing work. Data types allow programming languages to organize different kinds of data. Python has several native data types:

- Booleans (will be explained in this Chapter)
- Number will be explained in Step 6)
- String (will be explained in Step 7)
- List (will be explained in Step 9)
- Tuple (will be explained in Step 10)
- Sets (will be explained in Step 11)
- Dictionary (will be explained in Step 12)
- Bytes and byte arrays (will be explained in Step 24)

Every value in Python is an **object** with a **data type**. Hence, you’ll find data types like function, module, method, file, class, and compiled code.

Boolean Data Type

There are two built-in Boolean data types: **True** or **False**. These values are used in conditional expressions, comparisons, and in structures that require representation for truth or falsity.

To show how Boolean data types work, create three variables that will hold Boolean values resulting from the assigned expressions.

```
>>> bool_a = (12 >= 4)
>>> bool_b = (12 == 4*4)
>>> bool_c = (18 != 2*9)
```

Use the print function to see the values stored in each variable:

```
>>>print(bool_a, bool_b, bool_c)
```

You should now get : True False False. (see figure 12 below)

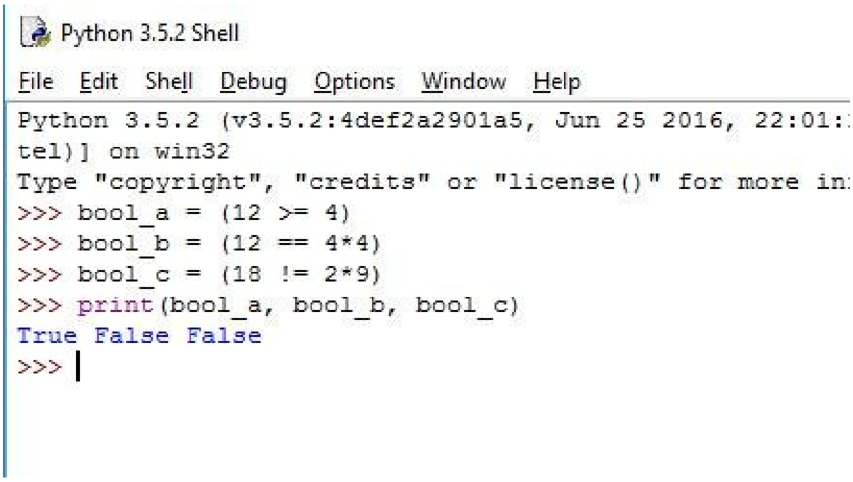


Figure 12.

Step 6: Number Data Types

There are **3 numeric data types** in Python 3: integers, floating point numbers, and complex numbers. Python can tell if the number is an integer or a float by the presence or absence of a decimal point. It recognizes a complex number by its standard form a + bJ. Hence, it's not necessary to declare a number as a specific type.

Integers (int)

Integers are whole numbers with no fractional parts and decimal point. They can be positive, negative, or zero. Integers can have **unlimited** size in Python 3 and are only limited by the available memory on your computer. Python supports normal integers as well as octal, hexadecimal, and binary literals:

Normal integers

Examples:

44

4500

-32

0

96784502850283491

Octal literal (base 8)

An octal literal is a number with a prefix of 0O or 00 (zero and uppercase letter O or lower case letter o).

Examples:

```
>>> ol = 0O40
>>>print(ol)
32
>>>
```

Enter 0O40 on the Python prompt to get the same integer equivalent:

```
>>>0O40
```

```
32
>>>
```

Hexadecimal literal (base 16)

A hexadecimal literal is a number with a prefix 0X or 0x (zero and uppercase or lowercase letter x).

Example:

```
>>>hl = 0XA0F
>>>print(hl)
2575
>>>
```

```
>>> 0XA0F
2575
>>>
```

Binary literal (base 2)

Binary literals are indicated by the prefix 0b (zero and uppercase or lowercase letter b).

Example:

```
>>> bl = 0b101010
>>> print(bl)
42
>>>
```

```
>>>0b101010
42
>>>
```

Converting Integers to their String Equivalent

With the **functions oct()**, **hex()**, and **bin()**, You can easily convert an integer to its string equivalent as needed.

To illustrate, convert the integer 45 to its octal, hexadecimal, and binary literal using the appropriate built-in functions:

integer to octal literal

>>> oct(45)

'0o55'

>>>

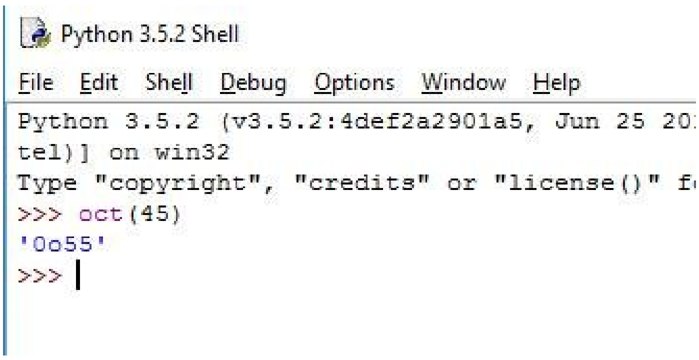


Figure 13. In this example, we have converted the integer 45 into it's octal literal equivalent O055.

integer to hexadecimal literal

>>>hex(45)

'0x2d'

>>>

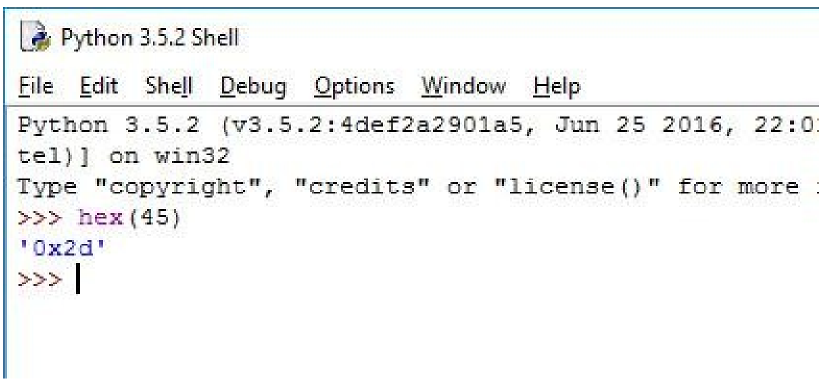


Figure 14.In this example, we have converted the integer 45 into it's hexadecimal literal

equivalent Ox2d.

integer to binary literal

>>>bin(45)

‘0b101101’

>>>

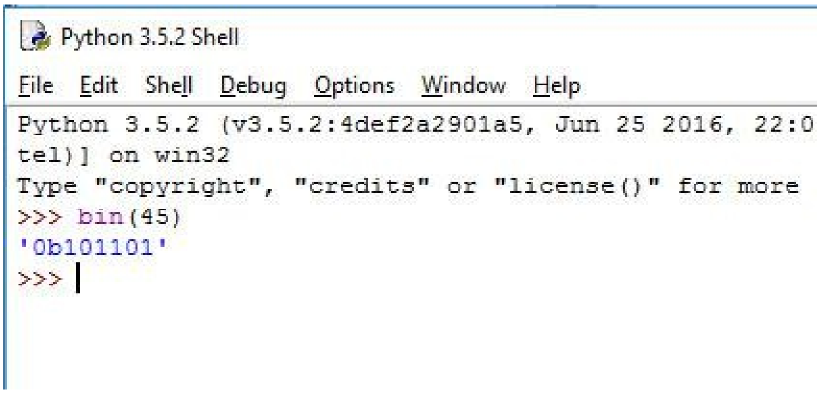


Figure 15. In this example, we have converted the integer 45 into it’s binary literal equivalent Ob101101.

Floating-Point Numbers (Floats)

Floating-point numbers are real numbers with a decimal point.

Examples:

0.50,

42.50

10500.40

2.5

Alternatively, floats may be expressed in scientific notation using the letter “e” to indicate the 10th power.

>>> 3.5e5

350000.0

>>>

```
>>> 1.2e4
12000.0
>>>
```

PLEASE NOTE

The letter “E” or “E notation” is used to express very large and very small results in scientific notation. Because exponents like 1023 cannot always be conveniently displayed, the letter E or e is used to represent 10 to the power of. So in the example above 3.5e5 means 3.5 x 105).

Complex Numbers

Complex numbers are pairs of **real** and imaginary **numbers**. It takes the form ‘a + bJ’ or ‘a + bj’ where the left operand is a **real number** while the right operand is a **float** (b) and an **imaginary number** (J). The upper or lower case letter Jj refers to the square root of -1, an imaginary number.

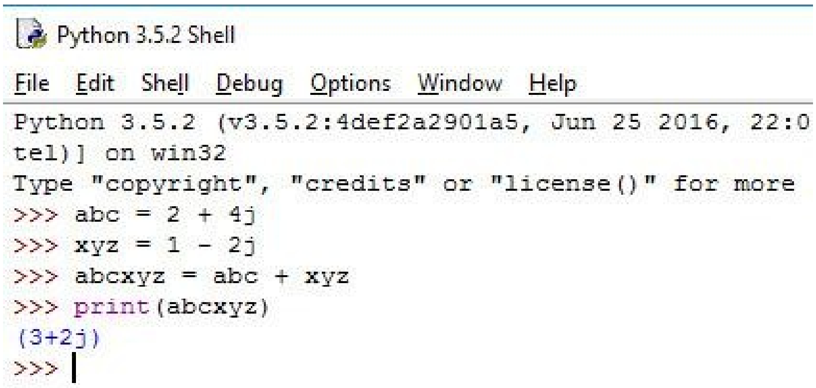
Real Number(a) **Float** (b) and an **Imaginary Number** (J).

a +	bJ
a +	bj

Example:

```
>>> abc = 2 + 4j
>>> xyz = 1 - 2j
>>> abcxyz = abc + xyz

>>>print(abcxyz)
(3+2j)
>>>
```



PLEASE NOTE

A **real number** is a value that represents a quantity along a line. The real numbers include all the rational numbers, such as the integer −3 and the fraction 2/3, and all the irrational numbers, such as √2.

An **imaginary number** is a complex number that can be written as a real number multiplied by the imaginary unit *i*.

Converting From One Numeric Type to Another

While Python is capable of converting mixed number types in an expression to a common type, you may need to explicitly convert one type to another to comply with the requirements of a function parameter or an operator. You can coerce Python to do this by using the **appropriate keyword** and **parameter** (the number to be converted).

To convert a float to a plain integer

Type int(x)

```
>>>int(97.50)
97
>>>
```

To convert an integer to a floating-point number

Type float(x)

```
>>>float(567)
567.0
>>>
```

To convert an integer to a complex number

Type complex(x)

```
>>>complex(34)
(34+0j)
>>>
```

To convert a float to a complex number

Type complex(x)

```
>>>complex(34.5)
(34.5+0j)
>>>
```

To convert a numeric expression (x,y) to a complex number with a real number and imaginary number

Type complex(x,y)

```
>>>complex(4,7)
(4+7j)
>>>
```

Take note that you cannot convert a complex number to an integer.

Python has many built-in operators that programmers can use to manipulate and utilize numeric data type.

Numbers and Arithmetic Operators

Python's arithmetic operators allow programmers to create useful applications and automate daily tasks. Python supports **7 arithmetic operators**:

Addition	+
Subtraction	-

Multiplication	*
Division	/
Exponent	**
Modulos	%
Floor Division	//

Addition (+)

The addition operator adds the operands on either side of the operator:

```
>>>20 + 14
34
>>>45.8 + 50.3
96.1
>>>2 + 3.5
5.5
>>> (2 + 3j) + (3 - 4j)
(5-1j)
```

Subtraction (-)

The subtraction operator subtracts the value of the right operand from the value of the left operand:

```
>>>50 - 7
43
>>>75.4 - 2.50
72.9
```

```
>>> 6 - 3.4
2.6
>>> (5 + 2j) - (4 - 6j)
(1+8j)
>>>
```

Multiplication (*)

The multiplication operator multiplies the operands on either side of the operator:

```
>>>4 * 3
12
>>> 8.4 * 2.5
21.0
>>> 8 * 2.5
20.0
>>>
```

Division (/)

The division operator divides the value of the left operand with the value of the right operand:

```
>>> 32/8
4.0
>>> 4.5 / 1.5
3.0
>>> 27.5 / 3.2
8.59375
>>>
```

Take note that the division operator always returns **a float by default** regardless of the data type of the operands. If you want your quotient to be in integer format, you can type **int()** and enclose the mathematical expression.

For example:

```
>>> int(32/8)
4
>>> int(4.5/1.5)
3
>>> int(27.5/3.2)
8
>>>
```

You can apply the same principle to the other arithmetic operations if you want Python to return a data type other than the default numeric type. For instance, if you need the expression to output an integer after entering an integer*float expression which results to a float, you can coerce Python to return an integer with int(). On the other hand, if you want Python to return a float, you can use float() and place the expression inside the parenthesis.

Exponent (**)

The exponent operator raises the first operand to the power indicated by the right operand:

```
>>> 3 ** 4
81
>>>4.5 ** 3
91.125
>>>
```

Modulos (%)

The modulos operator (%) returns the remainder after dividing the left operand with the right operand.

```
>>> 32%7
4
>>> 16.5%4.5
3.0
>>> 4%2.5
```

```
1.5
>>>
```

Relational or Comparison Operators

Comparison operators evaluate the expression and return either **True** or **False** to describe the relation of the left and right operands.

Python supports the following **relational operators**:

is less than	<
is greater than	>
is less than or equal to	<=
is greater than or equal to	>=
is equal to	==
is not equal to	!=

Examples:

```
>>> 15 == 3*2*2
False
>>>4*2<=5*3+2
True
>>> 15 != 3*5
False
>>> (8*4)> 24
```

```
True
>>> 10 < (2*12)
True
>>>25 >= (60*2)
False
>>>
```

Assignment Operators

Assignment operators are used to assign values to variables.

= Operator

The = operator assigns the value of the right operand to the left operand.

Examples:

```
>>>x = 2 * 4
>>>y = 2
>>>xy = x + y
>>>a, b, c = 10, 30.5, 15
>>>a = b = c = 100
```

add and +=

The add and (+=) operator adds the value of the left and right operands then assigns the sum to the left operand. It takes the format x+=a and its equivalent to the expression x = x+a.

Examples:

```
>>> x = 12
>>>y = 5
>>> y+=12
>>>print(y)
```

```
17
>>>
```

```
>>> a = 7
>>> b = 0
>>> b += a
>>>print(b)
7
>>>
```

```
>>> b = 0
>>> b += 5
>>>print(b)
5
>>>
```

subtract and -=

The subtract and (-=) operator first subtracts the value of the right number from the left, and then assigns the difference to the left number. It is expressed as x -= a and its equivalent expression is x = x-a.

Examples:

```
>>>a = 15
>>>b = 13
>>>a -= b
>>>print(a)
2
```

```
>>> counter = 10
>>> counter -= 5
>>>print(counter)
5
```

```
>>> counter = 0
>>> counter -= 10
>>>print(counter)
-10
```

multiply and *=

The multiply and (*=) operator takes the product of the left and right operands and assigns the value to the left operand. It is expressed as x*=a which is equivalent to x = x*a.

Examples:

```
>>> x = 15
>>> y = 3
>>> x *= y
>>>print(x)
45
```

```
>>> multiplier = 3
>>> multiplier *= 5
>>>print(multiplier)
15
```

divide and /=

The divide and (/=) operator first divides the value of the left operand with the right then assigns the result to the left operand. It is expressed with x /= a which is equivalent to the expression x = x/a.

Examples:

```
>>> a = 15
```

```
>>> xyz = 45
>>> xyz /= a
>>>print(xyz)
3.0
```

```
>>> amount = 12
>>> amount /= 5
>>>print(amount)
2.4
```

modulos and %=

The modulos and (%=) operator initially divides the left operand with the right and then assigns the remainder to the left operand. It is expressed as x %= a which is equivalent to the expression x = x % a.

Examples:

```
>>> x = 12
>>> y = 5
>>> x %= y
>>>print(x)
2
```

```
>>> rem = 30
>>> rem %= 12
>>>print(rem)
6
```

floor division and //=

The floor division and (//=) operator first performs a floor division on the left operand then assigns the result to the left operand. It is expressed as x//=a which is equivalent to the

expression x = x//a.

Examples:

```
>>> a = 30
>>> b = 14
>>>a //= b
>>>print(a)
2

>>> floor = 15
>>>floor //= 4
>>>print(floor)
3
```

Bill Calculator



Now that you’re familiar with variables and arithmetic operators, you’re ready to create a simple program that will help you automate the task of computing for your **restaurant bill**. Assuming your total bill covers the **basic meal cost**, **sales tax**, and **tip**, you will need the following information to start working on your program.

basic meal cost	?
sales tax rate	7% of basic meal cost
tip	20% of the sum of the basic meal cost and sales tax

First, you'll have to set up the variables to store the required data.

```
meal = 45
sales_tax = 7/100
tip = 20/100
```

Since the tip is based on the combined amount of the basic meal cost and sales tax, you'll have to pass the total to a variable by creating a new variable or simply reassigning **meal** to hold the new value. Assuming you have decided to simply reassign meal, you can use the '**add and**' **operator** to compute for the total value of the basic meal cost and sales tax.

```
meal += meal * sales_tax
```

Now that the variable meal contains the combined value of the meal and sales tax, you're ready to compute for the tip. You can create a new variable, bill, to store the total bill amount.

```
bill = meal + meal * tip
```

Here is the code for your bill calculator:

```
meal = 45
sales_tax =  7/ 100
tip = 20 / 100
meal += meal * sales_tax
bill = meal + meal * tip
```

REMEMBER

As explained in Step 2, in order to create a program you must; open IDLE -> click File -> New File -> copy the code for your bill calculator specified above -> go to File -> Save As -> save the program as calculator.py or any other name you want -> click Run -> click on Run Module.

To get the total bill amount, just type the word bill on the command prompt:

```
>>> bill
57.78
>>>
```

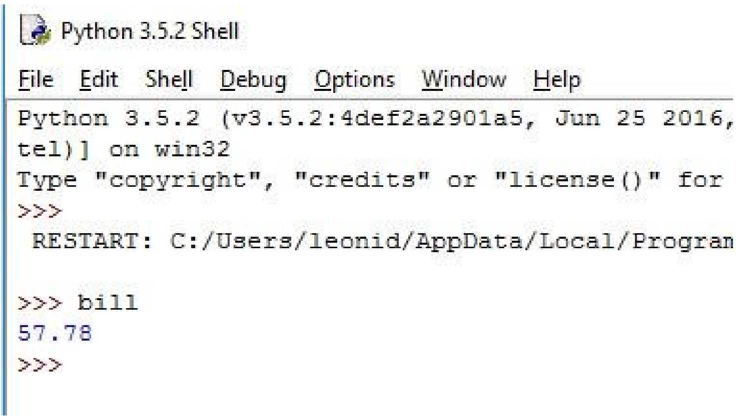


Figure 16. Type the word bill to get the total bill amount.

Built-in Functions Commonly Used with Numbers

abs(x)

Returns the absolute value of a given number

```
>>>abs(-4)
4
```

max()

Returns the largest in the given arguments. The **syntax** is:

max(x, y, z)

```
>>> max(34, 76, 5, -187, 95)
```

95

>>>print ("Maximum (50, 100, 150, 200) : ", max(50, 100, 150, 200))

Maximum (50, 100, 150, 200) : 200

>>>

min()

Returns the smallest argument. The **syntax** is:

```
min( x, y, z )
```

>>>min(65, 32, 65.5, -4, 0)

-4

>>>

round()

Returns a rounded number . The **syntax**is*:*

```
round(x[,n])
```

x is the number to be rounded

n is the number of digits from the decimal point

>>>round(104.987543, 2)

104.99

>>>round(99.01934567, 5)

99.01935

>>>

When the number of decimal places is not given, it defaults to zero.

>>>round(99.012345)

99

```
>>>
```

Math Methods

The following math methods are accessible after importing the math module.

PLEASE NOTE

Modules and how to import them will be explained later in Step 18.

To import math:

```
>>>import math
```

Math.ceil(x)

Returns the smallest integer which is not less than the given number.

Examples:

```
>>>import math
>>> math.ceil(98.75)
99
>>> math.ceil(154.01)
155
>>> math.ceil(math.pi)
4
>>>
```

Math.floor(x)

Returns the largest integer which is not greater than the given number.

Examples:

```
>>>import math
>>> math.floor(98.01)
98
>>> math.floor(13.99)
```

```
13
>>> math.floor(math.pi)
3
>>>
```

Math.fabs()

Returns the absolute value of a given number.

The fabs() method works similarly as the function abs() with some differences:

- fabs() works only on integers and floats while abs() works on integer, floats, and complex numbers
- fabs() returns a float while abs() returns an integer
- fabs() is imported from the math module while abs() is a built-in function

Examples:

```
>>>import math
>>> math.fabs(-12)
12.0
>>> math.fabs(15.0)
15.0
>>>
```

Math.pow()

Returns the value of x^y.

Examples:

```
>>>import math
>>>math.pow(5, 2)
25.0
>>> math.pow(2, 3)
8.0
>>> math.pow(3, 0)
```

```
1.0
>>>
```

Math.sqrt()

Returns the square root of a number which is larger than zero.

```
>>>import math
>>> math.sqrt(16)
4.0
>>> math.sqrt(25.0)
5.0
>>>
```

Math.log()

Returns natural logarithm of numbers greater than zero.

```
>>>import math
>>> math.log(100.12)
4.6063694665635735
>>> math.log(math.pi)
1.1447298858494002
>>>
```

Step 7: Strings

A string is an ordered series of Unicode characters which may be **a combination of one or more letters, numbers, and special characters**. It is an immutable data type which only means that once it is created, you can no longer change it.

To create a string, you must enclose it in either single or double quotes and assign it to a variable. For example:

```
>>>string_one = ‘a string inside single quotes’
>>>string_double = “a string enclosed in double quotes”
```

When you enclose a string which contains a single quote or an apostrophe inside single quotes, you will have to escape the single quote or apostrophe by placing a backslash (\) before it.

For example, to create the string, ‘I don’t see a single quote’:

```
>>> string_single = ‘I don't see a single quote.’
>>>
```

When you use the print function to print string_single, your output should be:

```
>>>print(string_single)
I don’t see a single quote.
>>>
```

Similarly, you will have to escape a double quote with a backslash (\) when the string is enclosed in double quotes.

Hence, to create the string = “He said: “You have been nominated as honorary president of the Mouse Clickers Club.””:

```
>>>string_two = “He said: "You have been nominated as honorary president of the Mouse Clickers Club. "”
>>>
```

```
>>>print(string_two)
He said: “You have been nominated as honorary president of the Mouse Clickers Club. “
>>>
```

Likewise, a backslash within a string should be escaped with another backlash.

```
>>> string_wow = “This is how you can escape a backlash \.”

>>>print(string_wow)
This is how you can escape a backlash .
>>>
```

Accessing Characters in a String

You can access individual characters in a string through indexing and a range of characters by slicing.

String Indexing

- The **initial character** in a string **takes zero as its index number** and the succeeding characters take 1, 2, 3… and so on as index numbers.
- To access the string backwards, the **last character** takes **-1 as its index**.
- A space is also considered a character.

To illustrate indexing in strings, define a variable named “string_var” and assign the string “Python String” with the statement:

```
>>>string_var = “Python String”
```

Index #	0	1	2	3	4	5	6	7	8	9	10	11	12

String	P	y	t	h	o	n		S	t	r	i	n	g
Index #	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Example #1:

To access the **first character** on the variable string_var (the first character of Python String is “P”), enter the variable name “string_var” and enclose the integer zero (0) inside the index operator or square brackets [].

```
>>> string_var[0]
‘P’
>>>
```

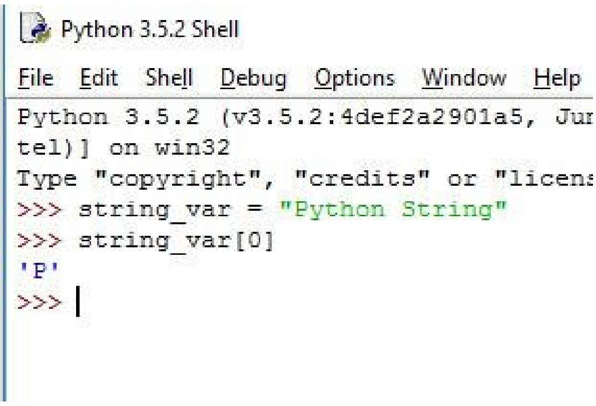


Figure 16. In this example, the first character of the string “Python String” is “P”. Since the first character takes zero as it’s index number, Python gives you the letter “P” as an answer.

Example #2:

To access the character on index 8, simply enclose 8 inside the square brackets:

```
>>> string_var[8]
‘t’
>>>
```

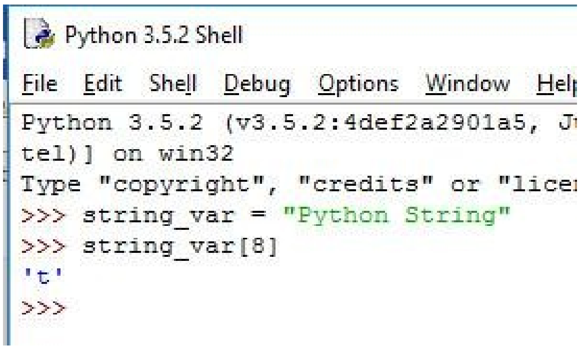


Figure 17. Since “t” takes 8 as it’s index number, Python gives you the letter “t” as an answer.

Example #3:

To access the character on index 6, an empty space:

```
>>> string_var[6]
‘ ‘
>>>
```

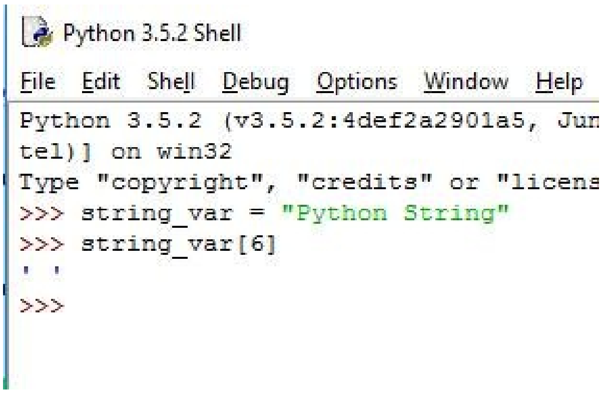


Figure 18. Since an empty space takes 6 as it’s index number, Python gives you ‘ ‘ (a space) as an answer.

Example # 4:

To access the last character of the string, you can use negative indexing in which the last character takes the -1 index.

```
>>>string_var[-1]
‘g’
>>>
```

A string is an ordered list, so you can expect that the penultimate letter takes the -2 index and so on.

Hence, -5 index is:

```
>>> string_var[-5]
't'
>>>
```

The Len() Function

There is a more sophisticated way of accessing the last character and it will prove more useful when you're writing more complicated programs: the **len() function**.

The len() function is used to determine **the size of a string, that is, the number of characters in a string.**

For example, to get the size of the variable 'string_var', you'll use the syntax:

```
>>>len(string_var)
13
>>>
```

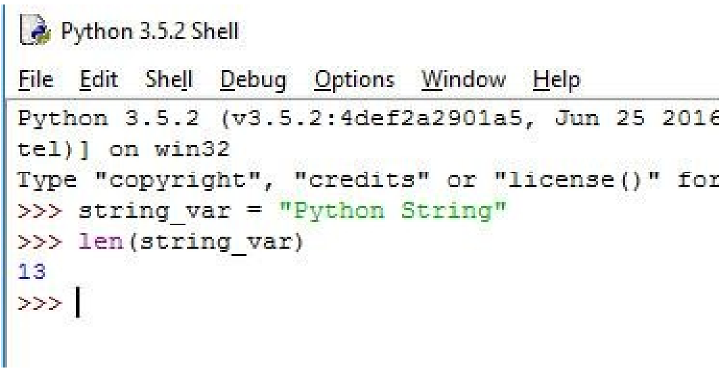


Figure 19. By using the len() function, Python is able to calculate the number of characters in your string "Python String". Do not forget that Python calculates a space as a character. Thus you get 13 characters.

Since the last character in the string takes an index which is one less than the size of the string, you can access the last character by subtracting 1 from the output of the len() function.

To illustrate, type the following on the command prompt:

```
>>> string_var[len(string_var)-1]
'g'
>>>
```

Some **important notes** about accessing strings through indexing:

- Always use an integer to access a character to avoid getting TypeError.
- Attempting to access a character which is out of index range will result to

an IndexError.

Slicing Strings

You can access a range of characters in a string or create substrings using the range slice **[:]** operator. To do this interactively on a random string, simply type the string within single or double quotes and indicate two indices within square brackets. A colon is used to separate the two indices. The slice operator will give you a string starting with S[A] and ending with S[B-1].

The syntax is: S[A:B-1]

S: The string you wish to use

A: The starting character of the substring you want to create

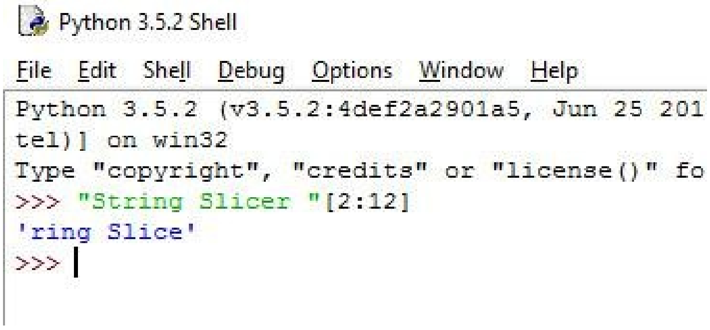
B: The ending character of the substring you want to create

Examples #1:

```
>>>"String Slicer "[2:12]
'ring Slice'
>>>
```

Index #	0	1	2	3	4	5	6	7	8	9	10	11	12

String	S	t	r	i	n	g		S	l	i	c	e	r
Index #	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 201
tel)] on win32
Type "copyright", "credits" or "license()" fo
>>> "String Slicer "[2:12]
'ring Slice'
>>> |
```

Figure 20. In this example, we want to create a substring using the range of characters from “r” to “e”.

Let’s use the syntax S[A:B-1] to understand this command;

- S: refers to the string “**String Slicer**”.
- A: **2** is the index number which is associated with the letter “r”.
- B-1: 12-1= **11** is the index number which is associated with the letter “ e”

Therefore, the command says that we are looking for a substring which includes the characters from “r” to “e” in the “String Slicer” string. Therefore, the answer is “ring Slice”.

Example #2:

>>>“Programmer”[3:8]

‘gramm’

>>>

Index #	0	1	2	3	4	5	6	7	8	9
String	P	r	o	g	r	a	m	m	e	r

Index #	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

You can also slice a string stored in a variable by performing the slicing notation on the variable using the following statements:

Example #3:

```
>>>my_var = “Python Statement”
>>> my_var[0:12]
‘Python State’
>>>
```

Index #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
String	**P**	**y**	**t**	**h**	**o**	**n**		**S**	**t**	**a**	**t**	**e**	**m**	**e**	**n**	**t**
Index #	-16	-15	-14	-12	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Example # 4:

```
>>>my_var = “Python Statement”
>>> my_var[7:11]
‘Stat’
>>>
```

When slicing or creating a substring, you can drop the first number if the starting character of the substring is the same as the initial character of the original string.

For example:

```
>>> test_var = “appendix”
```

```
>>>test_var [:6]
‘append’
>>>
```

Index #	0	1	2	3	4	5	6	7
String	**a**	**p**	**p**	**e**	**n**	**d**	**i**	**x**
Index #	-8	-7	-6	-5	-4	-3	-2	-1

In this example;

- The starting character is “a”, as is the first letter in “appendix”.
- The starting character of the substring is also “a”. Therefore, we can drop the first number. In fact, instead of writing **test_var[0:6]** you can simply write **test_var[:6].**

Similarly, if your substring ends on the last character of the string, you can drop the second index to tell Python that your substring ends on the final character of the original string.

```
>>> test_var = “appendix”
>>>test_var[3:]
‘endix’
>>>
```

Concatenating Strings

Several strings can be combined into one large string using the **+ operator.** For example, to concatenate the strings “I”, “know”, “how”, “to”, “write”, “programs “, “in”, “Python”, you can type:

```
>>>“I” + “know” + “how” + “to” + “write” + “programs” + “in” + “Python.”
>>>
```

You should get this output:

'IknowhowtowriteprogramsinPython.'

Likewise, you can concatenate strings stored in two or more variables. For example:

```
>>> string_one = "program"
>>> string_two = "is"
>>> string_three = "worth watching"
>>> print("An excellent "+string_one[:7] +" "+ string_two + " " + string_three +".")
>>>
```

When you run the program (when you press enter), the output will be:

An excellent program is worth watching.

Take note that since a string is immutable, the acts of slicing and concatenating a string do not affect the value stored in the variable.

For example, assign the string "concatenate" to the variable same_string and slice it to return the characters from index 4 to 6:

```
>>> same_string = "concatenate"
>>> same_string[4:7]
'ate'
>>>
```

Now, print the value on the variable same_string:

```
>>>print(same_string)
concatenate
>>>
```

Notice how the slicing did not affect the original string at all.

Repeating a String

To repeat a string or its concatenation, you'll use the **operator *and a number**. This instructs Python to repeat the string a certain number of times.

For example, if you want to repeat the string *<>* five times, you can type the string on the command prompt and specify the number of times it should be repeated with *5.

```
>>>“*<>*” *5
```

Here’s the output:

```
>>>‘*<>**<>**<>**<>**<>*’
```

You can store the above string in a variable and apply the * operator on the variable to achieve the same result:

```
>>>sign_string = “*<>*”
>>>sign_string * 5
‘*<>**<>**<>**<>**<>*’
>>>
```

Using the upper() and lower()functions on a string

The upper() and lower() functions can be used to print the entire string in uppercase or lowercase.

To illustrate, define a new variable ‘smart_var’ and use it to store the string “Europe”.

```
>>>smart_var = “Europe”
```

To print the entire string in **uppercase** letters, simply type:

```
>>>print(smart_var.upper())
```

The screen should display this output:

```
EUROPE
```

To turn things around, print the entire string on **lowercase** by typing:

```
>>>print(smart_var.lower())
```

You’ll get the output:

```
europe
```

The use of the upper() and lower() functions does not change the string stored at smart_var. You can prove this by entering the command:

```
>>>print (smart_var)
Europe
```

Using the str() function

You may sometimes need to **print non-string characters as string characters**. For example,a program may require you to print a string along with integers or other number types. Python’s **str() function**allows the non-string character to be converted to string characters.

To illustrate, you can create a variable to store the integer 246.The variable can then be used as parameter for the str() function.

```
>>>my_number = 246
>>>str(my_number)
‘246’
>>>
```

To print the string “My employee number is 246”, you can type the following:

```
>>>my_number = 246
>>> print(“My employee number is ” + str(my_number))
My employee number is 246
>>>
```

Python String Methods

There are several Python methods that can be used with string to support various operations:

The replace() method

The replace() method replaces a substring within an existing string with a new substring. Since you can't actually change the string on account of its immutable nature, replacing values necessitates the creation of a new string.

The **syntax** for the replace() method is:

```
source_string.replace(old_substring, new_substring, [ count])
```

Source_string: The string you want to perform the replacement in.

Old_substring: The existing substring in the source string you want to replace.

The new_substring: The old_substring will be replaced by new_substring.

Count: This count is an optional parameter. This specifies the maximum number of replacements. If not provided, the replace Python method will replace all occurrences. Also, the replacements start from the left if the count parameter is specified.

For example:

Let say you want to replace all occurrences of the substring 'flower', with the substring 'bug':

```
>>> x = "I saw a yellow flower in the garden, a yellow flower."
>>> y = x.replace('flower', 'bug')
>>>print(y)
I saw a yellow bug in the garden, a yellow bug.
>>>
```

Now let say you want to replace only the first occurrence of the substring, you can supply a third argument, 1:

```
>>> x = "I saw a yellow flower in the garden, a yellow flower."
>>> y = x.replace('flower', 'bug', 1)
>>>print(y)
I saw a yellow bug in the garden, a yellow flower.
>>>
```

Case Methods with String

There are four case methods that can be used with a string: upper(), lower(), swapcase(), and title(). These methods return a specific view but does not change the string itself.

Upper()

The upper() method returns a view of an entire string in **uppercase letters**.

```
>>> a = "garden"
>>> b = "a lovely garden"
>>> a.upper()
'GARDEN'
>>> b.upper()
'A LOVELY GARDEN'
>>>
```

Lower()

The lower() method returns a view of an entire string in **lowercase letters**.

```
>>> x = "HAPPY"
>>> y = "A HAPPY PLACE"
>>> x.lower()
'happy'
>>> y.lower()
'a happy place'
>>>
```

Swapcase()

The swapcase() method returns a view where the **lowercase letters are in uppercase and viceversa**.

```
>>> a = 'Garden'
>>> b = 'A Lovely Garden'
>>> a.swapcase()
'gARDEN'
>>> b.swapcase()
'a lOVELY gARDEN'
>>>
```

Title()

The title() method returns a view where the **first character of the string is capitalized** and **the rest are in lowercase**.

```
>>> x = "HAPPY"
>>> y = "A HAPPY PLACE"
>>> x.title()
'Happy'
>>> y.title()
'A Happy Place'
>>>
```

Count() method

This method is used to **sum up the occurrence of a given character or series of characters in a string**. It can be used to count words which are treated as a sequence of character.

For example:

```
>>> s = "It is not what you think it is but it is good enough for this purpose."
>>> s.count('is')
4
>>> s.count('t')
```

```
8
>>> s.count('it')
2
>>> s.count('not')
1
>>>
```

As you can see in this example, the occurrence "is" is found 4 times in this string.

"It is not what you think it is but it is good enough for this purpose."

The find() method

The find() method is used to **search for a given character or a sequence of characters in a string**.

Example #1:

```
>>> s = "A string is an immutable character or series of characters."
>>> s.find("string")
2
>>>
```

Index #	0	1	2	3	4	5	6	7
String	**A**		**s**	**t**	**r**	**i**	**n**	**g**
Index #	-8	-7	-6	-5	-4	-3	-2	-1

On the above example, the find() method returned ‘2’ which is the index of the first character of the string ‘string’.

Example #2:

On the following example, find() returns ‘5’, the index of the first occurrence of the string ‘i’.

```
>>> s = “A string is an immutable character or series of characters.”
>>> s.find(‘i’)
5
>>>
```

Example #3:

There are several i’s in the string and if you’re looking for the next ‘i’, you’ll have to supply a second argument which should correspond with the index immediately following index ‘5’ above. This tells the interpreter **to start searching from the given index going to the right**. Hence:

```
>>> s = “A string is an immutable character or series of characters.”
>>> s.find(‘i’, 6)
9
>>>
```

Example #4:

The search found the second occurrence of letter ‘i’ at index 9. Besides specifying an argument for the starting range, you can also provide an argument to indicate the **end to the search operation**. You can do it backwards by applying **negative indexing**. For example, if you want to find the third occurrence of the letter ‘i’, you can give the index ‘10’ as a second argument and provide an end to the search range with a third argument, -10.

```
>>> s = “A string is an immutable character or series of characters.”
>>> s.find(‘i’, 10, -10)
```

15

>>>

Isalpha()

The method isalpha() returns **True** if **all characters** of a non-empty string are **alphabetic**and there is at least one character and **False** if otherwise.

```
>>> s = ("programs")
>>> s.isalpha()
True
>>> print(s.isalpha())
True
>>> b = ("programming 1 and 2")
>>> b.isalpha()
False
>>>
```

As you can see the first 2 strings had only alphabetic characters (only letters) whereas the 3rd string had both alphabetic and numeric (letters and numbers) characters.

Isalnum()

The method isalnum() returns **True** if **ALL** the characters of a non-empty string are **alphanumeric** and False if otherwise

Example #1:

```
>>> b = "Programmer2"
>>> b.isalnum()
True
>>>
```

Example #2:

```
>>> a = "Programmer 1"
>>> a.isalnum()
False
>>>
```

As you can see, in the first example all characters are alphanumeric (characters which include both numbers and letters). However, the second example includes letters, numbers and **a space** between "programmer" and "1". This is why Python returns False.

Isidentifier()

The isidentifier() method tests for the validity of a given string as an identifier and returns True if valid or False if otherwise.

REMEMBER

As explained in Step 4, a Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Example #1:

```
>>> a = "program"
>>> a.isidentifier()
True
>>>
```

Example #2:

```
>>> b = "2program"
>>> b.isidentifier()
False
>>>
```

Example #3:

```
>>> c = "program_one"
```

```
>>> c.isidentifier()
True
>>>
```

Example # 4:

```
>>> d = "This is a string."
>>>d.isidentifier()
False
>>>
```

The join() method

The join() method returns a string in which the string elements of sequence have been joined by str separator. The first string or sequence of strings can be used as a separator as it is displayed as a trailing character after each character in the given sequence.

The **syntax** is:

```
str.join(sequence)
```

Sequence: This is a sequence of the elements to be joined.

Here is a sample code showing the usage of the join()method:

```
sep = "**";          # separator
seq = ("p", "r", "o", "g", "r", "a", "m", "m", "e", "r") # sequence of strings
print(sep.join(seq))
```

If you run the code, this should be your output:

```
p**r**o**g**r**a**m**m**e**r
```

REMEMBER

Read Step 2 to know how to run a program.

The characters in red are not part of the coding, they are simply comments. To read more about comments go to Step 4.

Lstrip() method

The lstrip method returns a copy of a string from which all leading characters or white spaces on the **LEFT** side of the string have been removed.

The **syntax** for the lstrip method is:

str.lstrip([chars])

Chars: You can supply what chars (characters) have to be trimmed.

The parameter allows you to instruct Python what character(s) should be trimmed.

Example # 1:

```
>>>str = "     this is a string with too many whitespaces";
>>>str.lstrip()
'this is a string with too many whitespaces'
>>>
```

Take note that this does not affect or trim the original string:

```
>>>str
'     this is string with too many whitespaces'
>>>
```

Example # 2:

```
>>> b = "******A special symbol is supposed to introduce this string.*****"
>>> b.lstrip('*')
```

‘A special symbol is supposed to introduce this string.*****’

>>>

If there is a leading whitespace and character that you want to remove at the same time, Python removes the leading whitespace even if you only specify the character as a parameter.

For example:

>>> b = “******A special symbol is supposed to introduce this string.*****”

>>> b.lstrip(‘*’)

‘A special symbol is supposed to introduce this string.*****’

>>> c = “   ******A special symbol is supposed to introduce this string.*****”

>>> b.lstrip(‘*’)

‘A special symbol is supposed to introduce this string.*****’

To use the **print function()** with the **lstrip() method**:

>>> x = “$$$$Money is not the most important thing in this world.”

>>> print(x.lstrip(‘$’))

Money is not the most important thing in this world.

>>>

Rstrip() method

The rstrip() method returns a string copy of a string in which all trailing characters on the **RIGHT** side of the string have been removed. By default, it strips the whitespaces at the end of a string, but you can supply the character you want to be trimmed as a parameter.

The **syntax** is:

str.rstrip([chars])

Chars: You can supply what chars (characters) have to be trimmed.

Example #1:

```
>>> s = “     This is an example of a string.     “
>>> s.rstrip()
‘     This is an example of a string.’
>>>
```

As you can see, Python has removed the whitespace at the end of the string. That is because we have not provided an argument and Python striped the whitespace at the end of the string by default.

Example #2:

```
>>>s = “This is an example of a string”
>>>s.rstrip(“g”)
>>>‘This is an example of a strin’
>>>
```

Here, we have remove “g” from the end of the string. This is because I have provided the letter “g” as the argument.

Strip([chars])

The strip() method does both lstrip() and rstrip() on a string and returns a copy of a string in which both leading and trailing characters have been removed.

Example #1:

```
>>> s = “     This string is in the middle of unnecessary whitespaces.     “
>>> s.strip()
‘This string is in the middle of unnecessary whitespaces.’
>>>
```

Example #2:

```
>>> y = “****I’m better off without these symbols.*****”
>>> y.strip(‘*’)
“I’m better off without these symbols.”
>>>
```

To use the **strip() method** with the **print()** function:

```
>>> y = “****I’m better off without these symbols.*****”
>>> y.strip(‘*’)
“I’m better off without these symbols.”
>>> print(y.strip(‘*’))
I’m better off without these symbols.
>>>
```

Rfind() method

Searches for the occurrence of a given substring

The rfind() method returns the index of a given substring’s **last occurrence** in the string and returns -1 if the substring is not found. It allows a start and end index arguments to limit the search range.

This is the **syntax** for rfind:

```
str.rfind(str, beg=0 end=len(string))
```

str: The string to be searched

beg: This parameter is optional. This is the starting index with a default of zero.

end: This parameter is optional. This is the ending index equal by default to the string’s length.

Example #1:

```
>>> str1 = "Programming is a skill that can be learned by anyone."
>>>str2 = "skill"
>>> str1.rfind(str2)
17
>>>
```

Python tells you that the substring "skill" is found on the17th index number.

Example #2:

```
>>> str1 = "Programming is a skill that can be learned by anyone."
>>>str2 = "skill"
>>> str1.rfind(str2, 0, 30)
17
>>>
```

Example #3:

```
>>> str1 = "Programming is a skill that can be learned by anyone."
>>>str2 = "skill"
>>> str1.rfind("be", 0, len(str1))
32
>>>
```

To use the **print() function** with **rfind():**

```
>>> str1 = "Programming is a skill that can be learned by anyone."
>>>str2 = "skill"
>>>print(str1.rfind(str2, 0, 25))
17
>>>
```

Index() method

Checks the occurrence of a given substring, returns index of the first occurrence.

The index() method checks the occurrence of a given substring in a string and returns the index of the first occurrence. It raises a ValueError if the substring is not found.

Its **syntax** is:

```
str.index(str2, beg=0 end=len(string))
```

str: The string to be searched

beg: This parameter is optional. This is the starting index with a default of zero.

end: This parameter is optional. This is the ending index equal by default to the string's length.

Examples:

In this example, you're going to instruct Python to search for the substring 'exam' without range parameters. The substring occurs twice in the string but the interpreter returns only its **first occurrence**:

Example #1:

```
>>> str1 = "This is a good example of a practical programming examination."
>>> str2 = "exam"
>>> str1.index(str2)
15
>>>
```

Assuming you want to find the index of the **next occurrence**, then you'll have to provide a **second parameter** to specify the starting index of the search. In this situation, since the initial occurrence of 'exam' is on index 15, you'll have to start the search at index 16.

Example #2:

```
>>> str1 = “This is a good example of a practical programming examination.”
>>> str2 = “exam”
>>> str1.index(str2, 16)
50
>>>
```

Example #3:

```
>>> str1 = “This is a good example of a practical programming examination.”
>>> str1.index(“ing”)
46
>>>
```

Example #4:

```
>>> str1 = “This is a good example of a practical programming examination.”
>>> print(str1.index(“am”, 47, len(str1)))
52
>>>
```

Python raises a ValueError if a substring is not found:

```
>>> str1 = “This is a good example of a practical programming examination.”
>>> str1.index(“bring”,5, len(str1))
Traceback (most recent call last):
  File “<pyshell#30>”, line 1, in <module>
    str1.index(“bring”, 5, len(str1))
ValueError: substring not found
>>>
```

Rindex() method

Searches for the occurrence of a substring, returns the index of the last occurrence.

This method returns the index of a substring's last occurrence in a given string or raises a ValueError if the substring is not found.

Its **syntax** is:

```
str.rindex(str, beg=0 end=len(string))
```

str: This specifies the string to be searched.

len: This is ending index, by default its equal to the length of the string.

beg: This parameter is optional. This is the starting index, by default its 0

end: This parameter is optional. This is the ending index equal by default to the string's length.

Examples:

```
>>> str1 = "Programming is a challenging and rewarding job."
>>> str2 = "ing"
```

To find the **last occurrence** of the substring "ing":

```
>>> str1.rfind(str2)
39
>>>
```

To tell Python to search the string for its **second occurrence backwards**, provide the index immediately before the first occurrence as the end parameter for rfind():

```
>>> str1 = "Programming is a challenging and rewarding job."
>>> str2 = "ing"
>>> str1.rfind(str2, 0, 38)
25
>>>
```

Finally, to find the **3rd occurrence** of the substring ‘ing’, supply 24 as the ending index:

```
>>> str1 = “Programming is a challenging and rewarding job.”
>>> str2 = “ing”
>>> str1.rfind(str2, 0, 24)
8
>>>
```

You will get the same output if you use the **find()** method instead:

```
>>> str1 = “Programming is a challenging and rewarding job.”
>>> str2 = “ing”
>>>str1.find(str2)
8
>>>
```

Zfill() method

Returns a string with leading zeroes within a specified width.

The zfill() method returns a string filled with leading zeros in the given width.

This is the **syntax** for zfil:

```
str.zfill(width)
```

width: Refers to the total width of the string.

Examples #1:

```
>>> str = ‘A03456’
>>>str.zfill(10)
```

'0000A03456'

>>>

Example #2:

>>> str = "A string may not be changed once it is created."

>>> str.zfill(60)

'0000000000000A string may not be changed once it is created.'

>>>

Rjust() method

Returns a right-justified string within a specified width.

The method returns a **right-justified string** based on the given width. It allows optional fill characters with space as default and returns the original string if the width is less than its length. Padding is done using the specified fillchar (default is a space)

The **syntax** is:

```
str.rjust(width[, fillchar])
```

width: This is the string length in total after padding.

fillchar: This is the filler character, default is a space.

Example #1:

>>> str = "Right justifying a string is sometimes necessary."

>>>str.rjust(70)

'   Right justifying a string is sometimes necessary.'

>>>

In this example, the original string is 49 index position long. Here, we do not specify the fillchar, therefore space is used as default.The Rjust method adds the amount of spaces needed to get the string 70 index position long (it added 21 spaces).

Example #2:

```
>>>str = “Right justifying a string is sometimes necessary.”
>>>str.rjust(70, ‘*’)
‘*********************Right justifying a string is sometimes necessary.’
>>>
```

In this example, the original string was 49 index position long. The Rjust method added the amount of “*” needed to getthe string 70 index position long (it added 21 “*”).

Example #3:

```
>>>str = “Right justifying a string is sometimes necessary.”
>>>print(str.rjust(55, ‘#‘))
######Right justifying a string is sometimes necessary.
>>>
```

Example #4:

However, if you specify a width that is less than the length of the string, Python simply returns the original string:

```
>>>str = “Right justifying a string is sometimes necessary.”
>>>print(str.rjust(30, ‘^’))
Right justifying a string is sometimes necessary.
>>>
```

Ljust() method

Returns a left-justified string within a specified width.

The ljust() returns a **left-justified** string based on the given width. It allows optional padding characters with space as default and returns the original string if the specified width is less than the actual width of the string.

The **syntax** is:

str.ljust(width[, fillchar])

width: This is the string length in total after padding.

fillchar: This is the filler character, default is a space.

Example #1:

```
>>>str = “Left justifying a string is cool.”
>>>str.ljust(45)
‘Left justifying a string is cool.            ‘
>>>
```

Example #2:

```
>>>str = “Left justifying a string is cool.”
>>>str.ljust(50, ‘*’)
‘Left justifying a string is cool.*******************’
>>>
```

Example #3:

Python simply returns the original string if the specified width is less than the actual length of the string:

```
>>>str = “Left justifying a string is cool.”
>>>str.ljust(25, ‘*’)
‘Left justifying a string is cool.’
>>>
```

Center() method

Returns a center-justified string within a specified width.

The center() method returns a **center-justified string** based on a specified width. It allows optional padding characters with space as the default filler.

The **syntax** is:

```
str.center(width[, fillchar])
```

width: This is the string length in total after padding.

fillchar: This is the filler character, default is a space.

Example #1:

```
>>>str="Center-justifying highlights the importance of a string"
>>>str.center(65)
'     Center-justifying highlights the importance of a string.     '
>>>
```

Example #2:

```
>>>str="Center-justifying highlights the importance of a string"
>>>str.center(65, '*')
'*****Center-justifying highlights the importance of a string.****'
>>>
```

Example #3:

```
>>>str="Center-justifying highlights the importance of a string"
>>>print(str.center(65, '^'))
^^^^^Center-justifying highlights the importance of a string.^^^^
>>>
```

Example #4:

If you specify a width which is less than the actual length of the string, Python merely returns the original string:

```
>>>str="Center-justifying highlights the importance of a string"
>>>str.center(40, '*')
'Center-justifying highlights the importance of a string.'
>>>
```

Endswith() method

Checks if the string ends in a given suffix.

The endswith() method returns **True** if the string's ending matches the given suffix or returns **False** if otherwise. The method allows the optional limitation of matching range with restricting start and end indices.

The **syntax** is:

```
str.endswith(suffix[, start[, end]])
```

suffix: This could be a string or could also be a tuple of suffixes to look for.

start: The slice begins from here.

end: The slice ends here.

Example #1:

```
>>>str = "A string is a sequence of characters, right?"
>>> suffix = "right?"
>>>str.endswith("right?")
True
>>>
```

Example #2:

```
>>>str = "A string is a sequence of characters, right?"
>>> suffix = "sequence"
>>>str.endswith(suffix, 0, 22)
True
>>>
```

Example #3:

```
>>>str = "A string is a sequence of characters, right?"
>>> suffix = "sequence"
>>>str.endswith(suffix, 0, 11)
False
>>>
```

Startswith() method

Checks if the string ends in a given suffix.

The startswith() method returns **True** if the string's prefix matches the given prefix or returns **False** if otherwise. The method allows the optional limitation of matching range with restricting start and end indices.

The **syntax** is:

```
str.startswith(suffix[, start[, end]])
```

suffix: This could be a string or could also be a tuple of suffixes to look for.

start: The slice begins from here.

end: The slice ends here.

Example #1:

```
>>>str = "Strings are immutable but not invincible."
>>> prefix = "Strings"
>>>str.startswith(prefix)
True
>>>
```

Example #2:

```
>>>str = "Strings are immutable but not invincible."
>>>prefix = "String"
>>>str.startswith(prefix)
True
>>>
```

Example #3:

```
>>>str = "Strings are immutable but not invincible."
>>> prefix = "S"
>>>str.startswith(prefix, 3, 22)
False
>>>
```

Iterating Through a String

You can use a 'for loop' to iterate through a string. For example, to count the number of occurrence of the letter 'o' in a given string, you can create a simple loop:

```
count = 0
for letter in'Python Programming':
if(letter == 'o'):
       count += 1
```

```
print(count, “letters found”)
```

When you run the program, you should get this output:

```
2 letters found
>>>
```

IMPORTANT

In Step 16, we will study loops more extensively. A loop is a control structure that allows the repetitive execution of a statement or group of statements. Loops facilitate complicated execution paths.

Step 8: Output Formatting

Python offers many built-in functions that can be used interactively at the prompt. The **print() function** is one of the functions that you will use in common, everyday programming life. There are several formatting options that you can use with print() to produce more readable and interesting output than values separated by space.

An output may be directed to the screen or file. You now know that you can use either expression statements or the print() function to display output on the screen. Another way to write values to a file is with the **write() method** which will be discussed later in the book.

The print() function

The print() function displays data on the default output device, the screen.

The following expressions demonstrate the standard usage of the print() function:

Example 1:

```
>>>print('Programming is a challenging and rewarding activity.')
Programming is a challenging and rewarding activty.
>>>
```

Example 2:

```
>>> x = 100
>>>print('The value of a is', x )
The value of a is 100
>>>
```

Example 3:

```
>>> x = 3 * 2
```

```
>>> y = 4 * 4
>>> xy = x + y
>>>print(xy)
22
>>>
```

Using the str.format() method to format string objects

The str.format() method is used to format string objects and produce more readable output in Python.

This section will show the different ways you can use the format() method to control your output.

The first example uses curly braces {} as place holders for the values supplied as arguments in the str.format expression.

Example #1:

```
>>> a='Programming is a challenging and {} activity.'
>>> a.format('rewarding')
'Programming is a challenging and rewarding activity.'
>>>
```

By default, values are displayed in the order in which they are positioned as arguments in the format() expression. If this is your preferred arrangement, you can tell Python to output the default order by referencing to the values with empty curly braces in the print parameter.

Example #2:

```
>>> x= 'The value of a is {}, b is {}, and c is {}.'
>>> x.format('50','55','100')
'The value of a is 50, b is 55, and c is 100.'
>>>
```

Example 3 :

```
>>> a = 50; b = 55; c = 100
>>>print(‘The value of a is {}, b is {}, and c is {}.’.format(a, b, c))
The value of a is 50, b is 55, and c is 100.
>>>
```

However, if you want the values to appear in **an order that is different from their positional arrangement**, you will have to specify their **index** inside the curly braces. Indexing starts from zero. Hence, the first value has index o, the second one has index 1, and so on.

Example #1:

```
>>> x= ‘The value of a is {2}, b is {0}, and c is {1}.’
>>> x.format(‘50’,‘55’,‘100’)
‘The value of a is 100, b is 50, and c is 55.’
>>>
```

Index #	0	1	2
Value	50	55	100

Example #2:

```
>>>a = 50; b = 55; c = 100
>>>print(‘The value of c is {2}, a is {0}, and b is {1}.’.format(a, b, c))
The value of c is 100, a is 50, and b is 55.
>>>
```

Python 3.5 still supports string formatting using the sprint() style used in C programming. Hence, you can still use the following formatting structure:

Example #3:

```
>>>print('Item No.:  %8d, Price per piece:  %8.2f'% (2546, 155.6287))
Item No.:      2546, Price per piece:    155.63
>>>
```

The first value was formatted to print an integer with a space of 8 digits (%8d) so the output shows 4 leading spaces to match the actual 4 digits usage of the given value, 2546. The second value was formatted to print a float with 8 digits and 2 decimal places. Since the value given only took up 3 digits, 5 leading spaces were added to the output. In addition, the 4 decimal places were rounded off to two.

The above print statement uses the **string modulo% operator**. You can translate the same statement into Python 3's string format method by using curly braces and positional arguments. Here's how your code will look like when you use the **format() method** instead of the **string modulo % operator.**

```
>>>print('Item No.:  {0:8d}, Price per piece:  {1:8.2f}'.format(2546, 155.6287))
Item No.:      2546, Price per piece:    155.63
>>>
```

Take note that this time, you have to supply the **index** of the value inside the curly braces along with the numeric format.

You can produce the same output by using keyword parameters instead of the index:

```
print('Item No.:  {x:6d}, Price per piece:  {y:7.2f}'.format(x=9875, y=167.9685))
```

Other Formatting Options

The string format method can also be used with formatting options such as zero-padding and data justification. Here are the options supported by Python 3:

'<'

Left justifies the value within the specified space.

Example #1:

```
>>>"{0:<35s} {1:6.2f}".format('Pepperoni and Cheese', 7.99)
'Pepperoni and Cheese                 7.99'
>>>
```

In the above example, you provided 35 spaces for the first value and instructed Python to print a left-justified value within the given space.

Here are examples of **left-justified formatting** with the **print function**:

Example #2:

```
>>>print("{x:<25s} {y:6.2f}".format(x='Ham and Cheese', y=4.75))
Ham and Cheese              4.75
>>>
```

Example #3:

```
>>> a = 'Black Forest'
>>> a1 = 4.99
>>> b = 'Double Dutch'
>>> b1 = 4.50
>>>print("The price of {0:<25s} is {1:6.2f} while the price of {2:<25s} is {3:6.2f} ".format(a, a1, b, b1))
The price of Black Forest              is   4.99 while the price of Double Dutch              is   4.50
>>>
```

'>'

Right-justifies the value within the given space.

Example #1:

```
>>>"{0:>20s} {1:7.2f}".format('Ham and Cheese:', 4.99)
```

```
'    Ham and Cheese:    4.99'
>>>
```

Example #2:

```
>>>print("There are {b:5d} bottles of {a:>18s} on stock.".format(a='Ginger Beer', b=3))
There are     3 bottles of        Ginger Beer on stock.
>>>
```

'^'

Center-justifies the value within the given space.

Example #1:

```
>>>"{0:^18s} {1:6.2f} {2:^15s}".format('Elbow Catch:',6.99, 'per piece')
'   Elbow Catch:      6.99    per piece
>>>
```

Example #2:

```
>>>print("The average price of a {c:^8s} of {a:^18s} is {b:5.2f}.".format(a='pizza', b=2.50, c='slice'))
The average price of a  slice   of       pizza        is  2.50.
>>>
```

'0'

Enables padding of leading zero(s) for numbers when placed before the width field.

The number of leading zeros to be added to the numeric value will be based on the number of digits allocated for the numeric field and the number of digits taken up by the value given.

Example #1:

```
>>>print("The code for Butterfly Hinge is {:06d}.".format(5482))
The code for Butterfly Hinge is 005482.
>>>
```

Example #2:

```
>>> print("The code for Elbow Catch is {:08d}.".format(12))
The code for Elbow Catch is 00000012.
>>>
```

'='

Forces sign placement before zero-padding.

Example:

```
>>>print("Elbow Catch has a stock of {0:=08d} {1:^8s}.".format(-8, 'pieces'))
Elbow Catch has a stock of -0000008  pieces .
>>>
```

Step 9: Lists

Python supports sequence data types that allow operations such as **indexing**, **slicing**, **multiplying**, **adding**, **removing**, and **membership checking**. List is one of the most commonly-used sequences and it is a most flexible type. Basically, a list can contain any type (string, float, integer, etc.)and number of items. It can hold one data type or a combination of several data types. It can even hold a list as an item. A list is mutable. Hence you can add, delete, or modify its elements.

Creating a list is as easy as defining a variable to hold an ordered series of items separated by a comma. This time, you will use a square bracket to enclose the items.

You can create an empty list with this **syntax**:

```
my_list = []
```

You can build a list by assigning items to it using the following syntax:

my_list = [item1, item2, item3, item4]

Here are examples of lists:

List with integers

num_list = [0, 5, 10, 15, 50, 14]

List with strings

string_list = [“cat”, “dog”, “lion”, “tiger”, “zebra”]

List with mixed data types

mixed_list = [18, “console”, 35.5]

List with nested list

nested_list = [“keyboard”, 8.5, 6, [5, 1, 3, 4, 5.5, 2]]

Accessing Elements on a List

There are several ways to access elements on a list:

Indexing

Just like strings, you can access items on a list with the **index operator [].** The first item has an index of zero (0). Remember to always use an integer when indexing to avoid TypeError. Attempting to access a list element that is beyond the index range will result to an IndexError.

Example #1:

```
>>>school_list = ["Biology", "English", "Chemistry", "Sociology", "Algebra"]
>>>school_list[0]
'Biology'
>>>
```

Index #	0	1	2	3	4
String	Biology	English	Chemistry	Sociology	Algebra

Example #2:

```
>>>school_list = ["Biology", "English", "Chemistry", "Sociology", "Algebra"]
>>>school_list[0][5]
'g'
>>>
```

Index #	0	1	2	3	4	5	6

String	B	I	O	L	O	G	Y

Example #3:

```
>>>school_list = [“Biology”, “English”, “Chemistry”, “Sociology”, “Algebra”]
>>> school_list[3]
‘Sociology’
>>>
```

To access nested list, you will use **nested indexing**.

```
>>>nested_list = [“code”, 4, [1, 3, 5, 7, 9]]
>>> nested_list[0]
‘code’
>>> nested_list[0][1]
‘0’
>>> nested_list[0][3]
‘e’
>>> nested_list[2]
[1, 3, 5, 7, 9]
>>> nested_list[1]
4
>>> nested_list[2][0]
1
>>> nested_list[2][3]
7
>>>
```

Negative Indexing

Python supports **negative indexing** for sequence types like lists. The last item on the list takes the -1 index, the second to the last item has the -2 index, and so on.

Example:

```
>>> quick_list = ["s", "h", "o", "r", "t", "c", "u", "t"]
>>> quick_list[-1]
't'
>>> quick_list[-3]
'c'
>>> quick_list[-7]
'h'
```

String	s	h	o	r	t	c	u	t
Index #	-8	-7	-6	-5	-4	-3	-2	-1

Slcing Lists

The slicing operator, the colon (:), is used to access a range of elements on lists.

REMEMBER

As explained in Step 7, the syntax for slicing a string is S[A:B-1]

S: The string you wish to use

A: The starting character of the substring you want to create

B: The ending character of the substring you want to create

Example #1:

```
>>> hw_list = ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
>>> hw_list[0:5]
['H', 'e', 'l', 'l', 'o']
>>>
```

Example 2:

```
>>> hw_list = [‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>> hw_list[5:]
[‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>>
```

Example #3:

```
>>> hw_list = [‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>> hw_list[3:6]
[‘l’, ‘o’, ‘W’]
>>>
```

Example #4:

```
>>> hw_list = [‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>> hw_list[:-4]
[‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’]
>>>
```

Example #5:

```
>>> hw_list = [‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>> hw_list[:]
[‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’]
>>>
```

Adding Elements to a List

Lists are mutable and you can easily add an element or a range of elements on a list with the **append()** or **extend() method**.

The append() method is used to add a single item while the extend() method is used to append two or more items. Both methods are used to add items at the end of the original list.

Append() metod

The **syntax** for the append() method is:

list.appendix(obj)

obj: This is the object to be appended in the list.

Example:

```
>>>even = [2, 4, 6, 8, 10, 12]
>>> even.append(14)
>>> even
[2, 4, 6, 8, 10, 12, 14]
>>>
```

Here we have added 14 to the even list.

Extend() method

The **syntax** for the extend() method is:

list.extend(seq)

seq: This is the list of elements.

Example:

```
>>>even = [2, 4, 6, 8, 10, 12]
>>>even.extend([16, 18, 20, 22])
>>> even
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
>>>
```

Here we have added a list of number to the even list.

Changing Elements of a List

You can change an item or a range of items on the list with the use of the assignment **operator (=)** and the indexing **operator [].**

For example, assuming that someone made an encoding mistake and came up with the following list of even numbers:

```
>>>even = [1, 3, 5, 7, 9, 12]
```

To rectify the mistake in the values entered, you can change the first item on the list with:

```
>>> even[0] = 2
```

Now, type “even” to view the updated list:

```
>>> even
[2, 3, 5, 7, 9, 12]
```

Changing the values individually will take time. You can instead specify an index range to change several values at once:

```
>>>even = [1, 3, 5, 7, 9, 12]
>>> even[1:5] = [4, 6, 8, 10]
```

Type even to see the updated even list:

```
>>> even
[2, 4, 6, 8, 10, 12]
>>>
```

Concatenating and Repeating Lists

It is possible to combine two lists in Python with the **+ operator**. In addition, you can use the ***operator** to repeat a list a certain number of times.

Combine two lists in Python with the + operator

Example #1:

```
>>> animals = [‘dog’, ‘cat’, ‘giraffe’, ‘bear’]
>>> babies = [‘puppy’, ‘kitten’, ‘calf’, ‘cub’]
>>> animals + babies
[‘dog’, ‘cat’, ‘giraffe’, ‘bear’, ‘puppy’, ‘kitten’, ‘calf’, ‘cub’]
>>>animals + babies + animals
[‘dog’, ‘cat’, ‘giraffe’, ‘bear’, ‘puppy’, ‘kitten’, ‘calf’, ‘cub’, ‘dog’, ‘cat’, ‘giraffe’, ‘bear’]
>>>
```

Example #2:

```
>>> vowels = [‘a’, ‘e’, ‘i’, ‘o’, ‘u’]
>>> vowels + [1, 2, 3, 4, 5]
[‘a’, ‘e’, ‘i’, ‘o’, ‘u’, 1, 2, 3, 4, 5]
>>>
```

To repeat a string with the * operator

Example #1:

```
>>> [‘do’, ‘re’, ‘mi’]*3
[‘do’, ‘re’, ‘mi’, ‘do’, ‘re’, ‘mi’, ‘do’, ‘re’, ‘mi’]
>>>
```

Example #2:

```
>>>abc = [‘a’, ‘b’, ‘c’]*4
>>> abc
[‘a’, ‘b’, ‘c’, ‘a’, ‘b’, ‘c’, ‘a’, ‘b’, ‘c’, ‘a’, ‘b’, ‘c’]
>>>
```

Inserting Item(s)

Earlier, you have learned how to add item(s) to a list with append() and extend(). Both methods added items to **the end of the original list**. This time, you will learn how to insert an item **on your desired location** with the **insert() method**.

The insert() method

Here's the **syntax**for the insert() method:

list.insert(index, obj)

index: This is the Index where the object obj need to be inserted.

obj: This is the Object to be inserted into the given list.

For example, here's a list called numbers:

```
>>> numbers = [1, 2, 4, 8, 9,10]
```

You want to insert number 3 right after number 2 and it will take index 2. Hence, you will have to enter this expression:

```
>>> numbers.insert(2,3)
```

To view the updated list, type numbers on the Python prompt:

```
>>> numbers
[1, 2, 3, 4, 8, 9, 10]
>>>
```

If you want the numbers list to contain a sequential list of numbers from 1 to 9, you can insert a range of items into the empty slice of the list.

To complete the numbers list, you will have to squeeze in the numbers 5, 6, and 7.

>>>numbers[4:3] = (5, 6, 7)

The above expression makes use of the slicing operator where the first number is the intended index(4) of the first item to be inserted and the second number is the numbers of items (3) you want to insert.

Here's the updated numbers list:

>>>numbers

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>>

Removing or Deleting Items from a List

To remove an item from a list, you can use either the **remove()** or the **pop() method**. The remove() method removes the specified item while the pop() method removes an item at the specified index. To empty a list, you can use the **clear() method**.

The remove() method

Here's the **syntax** for the remove() method:

```
my_list.remove(obj)
```

obj: This is the Object to be removed from the given list.

To illustrate, create a list called numbers:

>>> numbers = [1, 3, 5, 6.5, 7, 7.5, 9]

Assuming you want the list numbers to contain only integer values, you can use remove() to remove the floats, 6.5 and 7.5, one by one.

>>>numbers.remove(6.5)

>>>numbers.remove(7.5)

>>> numbers

[1, 3, 5, 7, 9]

>>>

The pop() method

The pop() method is used to remove the item associated with the given index. If the index is not given, it removes and returns **the last element on the list**.

Here's the **syntax**:

list_name.pop(index)

index: This is the Index where the item need to be removed.

To see how the pop() method works, create list called my_list:

>>>my_list = ["apples", "oranges", "yellow", "peach", "pear"]

You can easily see that there's an odd item on my_list, the color "yellow" and it is located on index 2. You can use the pop() method to remove this specific item:

>>> my_list.pop(2)

'yellow'

>>>

To view the updated my_list:

>>> my_list

['apples', 'oranges', 'peach', 'pear']

>>>

Now use the pop() method but don't specify an index:

>>> my_list.pop()

‘pear’

>>>

The pop() method just removed and returned the last item on the list. Here’s the updated my_list:

>>>my_list

[‘apples’, ‘oranges’, ‘peach’]

>>>

The clear() method

You may also prefer to just empty your list. You can do this with the clear() method. Here’s the **syntax**:

list_name.clear()

Hence, to clear my_list:

>>>my_list = [“apples”, “oranges”, “yellow”, “peach”, “pear”]

>>>my_list.clear()

>>> my_list

[]

>>>

Using the keyword del

You can also use the **keyword del** to delete one or more items or even the entire list.

REMEMBER

To see the list of keywords used in Python you can go back to Step 4.

To delete one item, the **syntax** is:

del list_name[]

Example:

```
>>> my_list = [“L”, “I”, “A”, “B”, “I”, “L”, “I”, “T”, “Y”]
>>>del my_list[0]          #delete the first item
>>> my_list
[‘I’, ‘A’, ‘B’, ‘I’, ‘L’, ‘I’, ‘T’, ‘Y’]
>>>
```

To delete a range of items, the syntax is:

del list_name[:]

Example:

```
>>> my_list = [“L”, “I”, “A”, “B”, “I”, “L”, “I”, “T”, “Y”]
>>>del my_list[1:6]          #delete multiple items from index 1 to 6
>>> my_list
[‘L’, ‘I’, ‘T’, ‘Y’]
>>>
```

Yet another way to let Python know that you want to delete a range of items is by assigning an empty list to the slice of items you want to delete.

To demonstrate, create my_list:

```
>>> my_list = [1, 2, 3, 4, “O”, “D”, “D”, 5, 6, 7, 8]
```

Now, delete the strings on index 4 to 6 by replacing it with an empty list:

```
>>> my_list[4:7] = []
```

Here’s the updated my_list:

```
>>> my_list
[1, 2, 3, 4, 5, 6, 7, 8]
>>>
```

You can delete the entire items on the list by replacing the entire range of items with an empty space:

```
>>> my_list = [1, 2, 3, 4, “O”, “D”, “D”, 5, 6, 7, 8]
>>> my_list[:] = []
>>> my_list
[]
>>>
```

Sorting Items on a List

The sort() method

The sort() method is used to sort items of similar data type within the list. It sorts the items in an **ascending order**.

The **syntax** is:

list.sort()

Example #1:

```
>>> list_1 = [1, 7, 4, 9.5, 3, 6.5, 15, 0]
```

Now, use the sort method and print the list:

```
>>> list_1.sort()
```

```
>>>print(list_1)
[0, 1, 3, 4, 6.5, 7, 9.5, 15]
>>>
```

Example #2:

```
>>> list_2 = ["blue", "yellow", "green", "orange", "red", "purple", "white"]
>>> list_2.sort()
>>>print(list_2)
['blue', 'green', 'orange', 'purple', 'red', 'white', 'yellow']
>>>
```

The reverse() method

The items in list_1 and list_2 are now arranged in ascending order. To arrange the items in the **reverse or descending order**, you can use Python's reverse method.

The **syntax** is:

```
list.reverse()
```

Now, apply the reverse() method to list_1 and list_2:

```
>>>list_1.reverse()
>>>print(list_1)
[0, 15, 6.5, 3, 9.5, 4, 7, 1]
>>>list_2.reverse()
>>>print(list_2)
['yellow', 'white', 'red', 'purple', 'orange', 'green', 'blue']
>>>
```

Take note that the sort() method is only applicable to lists containing items of similar types. Python will flash a TypeError:unorderable types if you try to sort a mixed list of **strings and integers**.

Using the count() Method on Lists

The count() method

The count() method counts of how many times an object occurs in a list.

The **syntax** is:

```
list.count(obj)
```

obj: This is the object to be counted in the list.

Examples:

```
>>> my_numbers = [4, 3, 2, 9, 3, 5, 4, 9, 3]
>>> my_numbers.count(3)
3
>>> my_numbers.count(4)
2
>>> my_numbers.count(9)
2
>>> my_numbers.count(2)
1
>>>
```

Testing for Membership on a List

Membership operators can be used to test if an object is stored on a list. There are two types of membership operators: “in” and “not in”. Python returns either True or False after evaluating the expression.

Using the “in” and “not in” operators

Example with integers:

```
>>> my_numbers = [1, 9, 38, 15, 4, 20, 7, 10]
>>> 7 in my_numbers
True
>>> 35 in my_numbers
False
>>> 15 not in my_numbers
False
>>> 50 not in my_numbers
True
>>>
```

Example with letters:

```
>>> my_letters = [“a”, “e”, “x”, “z”, “b”, “d”, “h”, “f”]
>>> “a” in my_letters
True
>>>“v” in my_letters
False
>>> “v” not in my_letters
True
>>> “f” not in my_letters
False
>>>
```

Using Built-in Functions with List

Python’s built-in functions such as min(), max(), len(), and sorted() can be used with list to obtain needed value and execute various tasks.

The following are built-in functions that are most commonly applied to lists:

Len()

Returns the number of items on a list.

Example:

```
>>>my_list = [0, 5, 10, 15, 20, 25]
>>>len(my_list)
6
>>>
```

Max()

Returns the largest item on a list.

Example:

```
>>> my_list = [0, 5, 10, 15, 20, 25]
>>>max(my_list)
25
>>>
```

Min()

Returns the smallest item on a list.

Example:

```
>>> my_list = [0, 5, 10, 15, 20, 25]
>>>min(my_list)
0
>>>
```

Sum()

Returns the total of all items on a list

Example:

```
>>> my_numbers = [5, 10, 20, 15, 25]
>>>sum(my_numbers)
75
>>>
```

Sorted()

Returns a sorted list (but the list itself is not sorted).

Example:

```
>>> odd = [1, 9, 15, 11, 3, 7]
>>>sorted(odd)
[1, 3, 7, 9, 11, 15]
>>>
```

To view the actual list, type odd on the prompt:

```
>>> odd
[1, 9, 15, 11, 3, 7]
>>>
```

List()

Convert iterables (tuple, set, dictionary) to a list.

PLEASE NOTE

We will later discuss tuple, set and dictionary in Step 10, 11 and 12.

To convert a string to a list

Example #1:

```
>>>list("Programmer")
['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'e', 'r']
>>>
```

Example #2:

```
>>>string_new = ("Programmer")
>>>list(string_new)
['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'e', 'r']
```

>>>

To convert a dictionary to a list

Example #1:

```
>>>list({“Name”: “John”, “Age”:24, “Occupation”:“employee”})
[‘Occupation’, ‘Name’, ‘Age’]
>>>
```

Example #2:

```
>>> my_dict = {“Name”:“Randy”, “Age”: 17, “Rating”: 90}
>>>list(my_dict)
[‘Age’, ‘Rating’, ‘Name’]
>>>
```

To convert a tuple to a list:

Example #1:

```
>>>list((“stocks”, “bonds”, “currency”, “coins”))
[‘stocks’, ‘bonds’, ‘currency’, ‘coins’]
>>>
```

Example #2:

```
>>> my_tuple = (“budget”, 2016, “projection”, 2017, “hedging”, “research”)
>>> list(my_tuple)
[‘budget’, 2016, ‘projection’, 2017, ‘hedging’, ‘research’]
>>>
```

Enumerate()

Returns an enumerate object which contains the value and index of all list elements as tuple.

```
>>> new_list = [2, 4, 6, 8, 10, 12, 14, 16]
>>>enumerate(new_list)
<enumerate object at 0x039865D0>
>>>
```

List Comprehension

List comprehension is a concise way of creating a new list from an existing list. It consists of an expression and a ‘for statement’ enclosed in square brackets. To illustrate, here is an example of creating a list where each item is an increasing power of 3.

```
>>> pow3 = [3 ** y for y in range(15)]
>>> pow3
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147, 531441, 1594323, 4782969]
>>>
```

The above code is equivalent to:

```
>>>pow3 = []
>>>for y in range(15):
   pow3.append(3 ** x)
```

A list comprehension may have more ‘for’ or ‘if’ statements. An if statement can be used to filter out elements for a new list.

Example #1:

```
>>> pow3 = [3 ** x for x in range(15) if x > 5]
>>> pow3
[729, 2187, 6561, 19683, 59049, 177147, 531441, 1594323, 4782969]
>>>
```

Example #2:

```
>>> even = [x for x in range(18) if x % 2 == 0]
>>> even
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16]
>>>
```

Example #3:

```
>>> odd = [y for y in range(18) if y % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17]
>>>
```

Step 10: Tuples

A tuple is a sequence type that contains an ordered collection of objects. A tuple, unlike a list, is immutable; you won't be able to change its elements once it is created. A tuple can hold items of different types and can have as many elements as you want subject to availability of memory.

Besides being immutable, you can tell a tuple apart from a list by the use of parentheses instead of square brackets. The use of parentheses, however, is optional. You can create a tuple without them. A tuple can store items of different types as well as contain any number of objects.

How to Create a Tuple

To create a tuple, you can place the items within parentheses and separate them with a comma.

Example of a numeric tuple

mytuple_x = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)

Example of a mixed-type tuple

mytuple_y = (“soprano”, 10, 4.3)

Example of a string tuple

mytuple_z = (“b”, “Jon”, “library”)

It's likewise possible to create a **nested tuple**:

my_tuple4 = (“Python”, (5, 15, 20), [2, 1, 4])

You can create **a tuple with only one item** but since this will look like a string, you'll have to place a comma after the item to tell Python that it is a tuple.

my_tuple5 = (“program”,)

You may also create an **empty tuple:**

my_tuple = ()

You can create a tuple **without the parentheses**:

numbers = 5, 3, 4, 0, 9

Accessing Tuple Elements

There are different ways to access items in a tuple.

Indexing

If you know how to access elements in a list through indexing, you can use the same procedure to access items in a tuple. The index operator indicates the index of the element you want to access. The first element is on index zero. Accessing an item outside the scope of the indexed elements will generate an IndexError. In addition, accessing an index with a non-integer numeric type will raise a NameError.

To illustrate how tuples work, create my_tuple with strings as elements.

```
>>> my_tuple = (‘p’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’, ‘m’, ‘e’, ‘r’)
>>>
```

To access the first element on the tuple:

```
>>> my_tuple[0]
‘p’
>>>
```

To access the 8th element:

```
>>> my_tuple[7]
‘m’
>>>
```

To access the 6th element:

```
>>> my_tuple[5]
‘a’
```

```
>>>
```

Negative Indexing

As it is a sequence type, Python allows negative indexing on tuples. The last element has -1 index, the penultimate element has -2 index, and so on.

```
>>> my_tuple = (‘p’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’, ‘m’, ‘e’, ‘r’)
>>> my_tuple[-1]
‘r’
>>>my_tuple[-7]
‘g’
>>>
```

Slicing a Tuple

If you want to access several items at the same time, you will have to use the slicing operator, the colon **(:)**. By now, you must be very familiar with how slicing works.

To see how you can slice a range of items from a tuple, create new_tuple:

```
>>>new_tuple = (‘i’, ‘m’, ‘m’, ‘u’, ‘t’, ‘a’, ‘b’, ‘l’, ‘e’)
>>>
```

To access the elements on the 4th to the 6th index:

```
>>> new_tuple[4:7]
(‘t’, ‘a’, ‘b’)
>>>
```

4 is the index of the first item and 7 is the index of the first item to be excluded.

To access tuple elements from index 2 to the end:

```
>>> new_tuple[2:]
(‘m’, ‘u’, ‘t’, ‘a’, ‘b’, ‘l’, ‘e’)
>>>
```

To access tuple items from the beginning to the 3rd index:

```
>>> new_tuple[:4]
('i', 'm', 'm', 'u')
>>>
```

Changing, Reassigning, and Deleting Tuples

A tuple is**immutable** so you cannot alter its elements. However, if it contains an element which is a mutable data type, you can actually modify this particular element. This is true in situations where one of the elements is a list. In such cases, you can modify the nested items within the list element.

```
>>> my_tuple = ('a', 5, 3.5, ['P', 'y', 't', 'h', 'o', 'n'])
>>>
```

Replacing a Tuple

To replace the item on index 2 of the list which is on index 3 of my_tuple:

```
>>>my_tuple[3][2] = 'x'
>>>
```

3 is the index of the list, 2 is the index.

```
>>> my_tuple
('a', 5, 3.5, ['P', 'y', 'x', 'h', 'o', 'n'])
>>>
```

While you may not replace or modify other data types, you can reassign a tuple to an entirely different set of values or elements.

Reassigning a Tuple

To reassign a tuple, you can just list a different set of elements and assign it to the tuple. To reassign new_tuple:

```
>>> my_tuple = ('c', 'o', 'd', 'e', 'r')
>>>
```

Deleting a Tuple

To delete a tuple and all the items stored in it, you will use the **keyword del**.

The **syntax** is:

del tuple_name

Hence, to delete new_tuple:

```
>>>del my_tuple
```

Tuple Membership Test

To test if a tuple contains a specific item, you can use the membership operators ‘in’ and ‘not in’

```
>>> our_tuple = (‘p’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’, ‘m’, ‘e’, ‘r’)
>>>‘g’in our_tuple
True
>>>‘l’in our_tuple
False
>>>‘e’not in our_tuple
False
>>>‘x’not in‘our_tuple’
True
>>>
```

Python Tuple Methods

Only two Python methods work with tuples:

Count(x)

Returns the number of elements which is equal to the given element.

The **syntax** is:

```
mytuple.count(a)
```

Example:

```
>>> new_tuple = (“p”, “r”, “o”, “g”, “r”, “a”, “m”, “m”, “e”, “r”)
>>> new_tuple.count(‘m’)
2
>>> new_tuple.count(‘r’)
3
>>> new_tuple.count(‘x’)
0
>>>
```

Index(x)

Returns the index of the first element which is equal to the given element.

The **syntax** is:

```
mytuple.index(a)
```

Example:

```
>>> new_tuple = (“p”, “r”, “o”, “g”, “r”, “a”, “m”, “m”, “e”, “r”)
>>> new_tuple.index(‘m’)
6
>>> new_tuple.index(‘r’)
1
>>> new_tuple.index(‘g’)
3
>>>
```

Built-in Functions with Tuples

Several built-in functions are often used with tuple to carry out specific tasks. Here are the functions that you can use with a tuple:

Len()

Returns the number of elements on a tuple.

```
>>> tuple_one = (‘cat’, ‘dog’, ‘lion’, ‘elephant’, ‘zebra’)
>>>len(tuple_one)
5
>>>
```

Max()

Returns the largest element 0n a tuple.

```
>>> numbers_tuple = (1, 5, 7, 9, 10, 12)
>>>max(numbers_tuple)
12
>>>
```

When a tuple holds items of purely string data type, max() evaluates the items alphabetically and returns the last item.

```
>>> my_tuple = (‘car’, ‘zebra’, ‘book’, ‘hat’, ‘shop’, ‘art’)
>>>max(my_tuple)
‘zebra’
>>>
```

Using max() on tuples with mixed data types (string and numbers) will raise a TypeError due to the use of unorderable types.

Min()

Returns the smallest element on a tuple.

```
>>> numbers_tuple = (1, 5, 7, 9, 10, 12)
>>>min(numbers_tuple)
1
```

```
>>>
```

When used on a tuple that contains purely string data type min() evaluates the items alphabetically and returns the first item.

```
>>> my_tuple = ('car', 'zebra', 'book', 'hat', 'shop', 'art')
>>> min(my_tuple)
'art'
>>>
```

Sorted()

Returns a sorted list but does not sort the tuple itself.

```
>>> my_tuple = ('dog', 'bird', 'ant', 'cat', 'elephant')
>>> sorted(my_tuple)
['ant', 'bird', 'cat', 'dog', 'elephant']
>>>
```

The order of elements inside the my_tuple, however, remains the same:

```
>>> my_tuple
('dog', 'bird', 'ant', 'cat', 'elephant')
>>>
```

Sum()

Returns the total of all items on a tuple.

```
>>> my_tuple = (5, 10, 15, 20, 25, 30)
>>>sum(my_tuple)
105
>>>
```

Tuple()

Converts iterables like string, list, dictionary, or set to a tuple.

How to convert a string to a tuple

Example #1:

```
>>>tuple("Programmer")
('P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'e', 'r')
>>>
```

Example #2:

```
>>> my_string = ("Hello World")
>>>tuple(my_string)
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd')
>>>
```

How to convert a dictionary to a tuple

Example #1:

```
>>>tuple({'Name':'Joshua', 'Animal':'elephant', 'Color': 'blue', 'Age':22} )
('Age', 'Color', 'Name', 'Animal')
>>>
```

Example #2:

```
>>> my_dict = {'Name':'Jack', 'Area':'Florida', 'subscription':'premium'}
>>> tuple(my_dict)
('Name', 'Area', 'subscription')
>>>
```

How to convert a list to a tuple

Example #1:

```
>>>tuple(['red', 'blue', 'yellow', 'green', 'orange', 'violet'])
('red', 'blue', 'yellow', 'green', 'orange', 'violet')
```

```
>>>
```

Example #2:

```
>>> my_list = [‘interest’, ‘rate’, ‘principal’, ‘discount’, ‘rollover’]
>>>tuple(my_list)
(‘interest’, ‘rate’, ‘principal’, ‘discount’, ‘rollover’)
>>>
```

Enumerate()

Returns an enumerate object containing the value and index of all tuple elements as pairs.

```
>>> my_tuple = (1, 3, 5, 7, 9, 11, 13, 15)
>>>enumerate(my_tuple)
<enumerate object at 0x03237698>
>>>
```

Iterating through a Tuple

You can iterate through each item in a tuple with the **‘for’ loop**.

```
>>> for fruit in (‘apple’, ‘peach’, ‘pineapple’, ‘banana’, ‘orange’):
        print(“I love ” + fruit)

I love apple
I love peach
I love pineapple
I love banana
I love orange
```

Tuples vs. Lists

Except for the symbols used to enclose their elements and the fact that one is mutable and the other is not, tuples and lists are similar in many respects. You will likely use a tuple to hold elements which are of different data types while you will prefer a list when working on elements of similar data types.

There are good reasons to choose tuple over a list to handle your data.

The immutable nature of tuples results in faster iteration which can help improve a program’s performance.

Immutable tuple elements can be used as dictionary keys, something that is not possible with a list.

Implementing unchangeable data as a tuple will ensure that it will stay write-protected.

Step 11: Sets

A set is **an unordered group** of unique elements. Although a set itself is mutable, its elements must be immutable. Sets are used to carry out math operations involving sets such as intersection, union, or symmetric difference.

Creating a Set

You can create a set by enclosing all elements in curly braces {} or by using set(), one of Python's built-in functions. A set can hold items of different data types such as float, tuple, string, or integer. It cannot, however, hold a mutable element such as a dictionary, list, or set. Sets can hold any number of items. A comma is used to separate items from each other.

An example of s set of integers:

```
>>>my_set = {1, 4, 6, 8, 9, 10}
```

An example of a set of strings:

```
>>>my_set = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’}
```

An example of a set of mixed data types:

```
>>> my_set = {5.0, “Python”, (5, 4, 2), 6}
```

An example of a set created from a list:

```
>>>set([5,4,3, 1 ])
{1, 3, 4, 5}
>>>
```

A set's elements cannot have a duplicate. When you create a set, Python evaluates if there are duplicates and drops the duplicate item.

Example #1:

```
>>> my_set = {'apple', 'peach', 'grape', 'apple', 'strawberry', 'grape'}
>>>my_set
{'peach', 'grape', 'apple', 'strawberry'}
>>>
```

Example #2:

```
>>>{1, 3, 5, 9, 1, 4, 3}
{1, 3, 4, 5, 9}
>>>
```

```
>>>set(['a', 'c', 'd', 'a', 'g', 'h', 'd'])
{'a', 'c', 'h', 'g', 'd'}
>>>
```

To create an empty set, you will have to use the set() function without an argument. You cannot use empty curly braces as this is the syntax for creating an empty dictionary.

```
>>> my_set = set()
>>>type(my_set)
<class 'set'>
>>>
```

Changing Elements on a Set

Sets are mutable so you can change their elements. Because sets are unordered, you cannot access or change an item or items through indexing or slicing like what you did earlier with strings and lists. You can, however, change the elements of a set with the methods **add()** or **update()**. The method add() appends a single element to a set while update() adds multiple elements. Strings, lists, tuples, or other sets can be used as argument when you use the update() method.

Example #1:

```
>>> my_set = {2, 4, 6, 8, 10}
>>> my_set.add(12)
>>> my_set
```

```
{2, 4, 6, 8, 10, 12}
>>>
```

Example #2:

```
>>> my_set = {2, 4, 6, 8, 10}
>>> my_set.update([14, 16, 18, 20])
>>> my_set
{2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
>>>
```

Example #3:

```
>>> my_set = {2, 4, 6, 8, 10}
>>> my_set.update('a', 'b')
>>> my_set
{2, 'a', 4, 6, 8, 10, 'b'}
>>>
```

Removing Set Elements

The **remove()** and **discard()** methods can be used to remove a specific item from a set. The only difference between the two methods is their response to a non-existent argument. The use of the remove() method raises an error when the item given as argument does not exist. With discard(), the set simply remains unchanged.

Example #1:

```
>>> my_set = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> my_set.discard('c')
>>> my_set
{'b', 'e', 'f', 'g', 'a', 'd'}
>>>
```

Example #2:

```
>>> my_set = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> my_set.remove('f')
>>> my_set
{'e', 'd', 'b', 'c', 'a', 'g'}
>>>
```

Here is how Python responds when you use discard() with an item which is not foundon the set:

```
>>> my_set = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> my_set.discard('x')
>>> my_set
{'g', 'd', 'e', 'c', 'a', 'f', 'b'}
>>>
```

The elements on the set were unchanged and no error was raised.

On the other hand, remove() will raise a Key Error if the item given as an argument is non-existent:

```
>>> my_set = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
>>> my_set.remove('x')
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    my_set.remove('x')
KeyError: 'x'
>>>
```

The **pop()** method is likewise used to remove and return an item on a set. Since the **set is unordered**, you cannot possibly control which item will be popped. Selection is arbitrary.

```
>>> my_set = {'a', 'e', 'i', 'o', 'u'}
>>> my_set.pop()
```

```
'o'
>>> my_set
{'e', 'a', 'i', 'u'}
>>>
```

Finally, you can use the **clear()** method to remove all elements on a set.

```
>>> my_set = {'a', 'e', 'i', 'o', 'u'}
>>> my_set.clear()
>>> my_set
set()
>>>
```

Set Operations

You can use sets to perform various set operations. To do this, you will use different Python operators or methods.

Set Union

A union of two sets refers to a set that contains all elements from the given sets. You can use the **| operator** or the **union()** method to perform the operation. The result is a combination of all elements which are returned in an **ascending order.**

Example with the | operator:

```
>>> X = {1, 3, 5, 7, 9, 11, 13}
>>> Y = {2, 4, 6, 8, 10, 12, 14}
>>> X | Y
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
>>>
```

Example with the union() method:

```
>>> X = {1, 3, 5, 7, 9, 11, 13}
>>> Y = {2, 4, 6, 8, 10, 12, 14}
>>> X.union(Y)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
>>>
```

Or

```
>>> X = {1, 3, 5, 7, 9, 11, 13}
>>> Y = {2, 4, 6, 8, 10, 12, 14}
>>> Y.union(X)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
>>>
```

Set Intersection

The intersection of two sets refers to the **set of common elements between them**. It is performed with either the **& operator** or the **intersection() method**. Both return a set with elements that are arranged in **ascending order**:

Example with the & operator:

```
>>> x = {1, 3, 5, 7, 2, 4, 6}
>>> y = {2, 4, 5, 7, 2, 0, 9}
>>> x & y
{2, 4, 5, 7}
>>>
```

Example with the intersection() method:

```
>>> x = {1, 3, 5, 7, 2, 4, 6}
>>> y = {2, 4, 5, 7, 2, 0, 9}
>>> x.intersection(y)
{2, 4, 5, 7}
>>> y.intersection(x)
{2, 4, 5, 7}
>>>
```

Set Difference

Set difference refers to a set of elements that are found in one set but not in the other set. For instance, the difference of X and Y (X – Y) is a set of elements that can be found in X but not in Y. Conversely, the difference of Y and X (Y – X) is a set of elements that are found in Y but not in X. The set difference operation is performed with either the **– operator** or the **difference() method**.

Examples with the - operator:

```
>>> x = {1, 2, 3, 5, 7, 9}
>>> y = {2, 8, 9, 5, 2, 1}
>>> x - y
{3, 7}
>>> y - x
{8}
>>>
```

Examples with the difference() method:

```
>>> x = {1, 2, 3, 5, 7, 9}
>>> y = {2, 8, 9, 5, 2, 1}
>>> x.difference(y)
{3, 7}
>>> y.difference(x)
{8}
>>>
```

Set Symmetric Difference

The symmetric difference between two sets refers to the set of elements that are not common in both sets. It is performed with either the ^ **operator** or the **symmetric_difference() method**.

Example with the ^ operator:

```
>>> a = {1, 3, 5, 4, 6, 8}
>>> b = {5, 2, 6, 1, 8, 10}
>>> a ^ b
{2, 3, 4, 10}
>>>
```

Examples with the symmetric_difference() method:

```
>>> a = {1, 3, 5, 4, 6, 8}
>>> b = {5, 2, 6, 1, 8, 10}
>>> a.symmetric_difference(b)
{2, 3, 4, 10}
>>> b.symmetric_difference(a)
{2, 3, 4, 10}
>>>
```

Set Membership Test

The **membership test operators**(the “in” and “not in” operators) can be used to test the existence or non-existence of an item on a set.

For example:

```
>>> my_set = {‘land’, ‘sea’, ‘air’, ‘ocean’, ‘river’}
>>>‘sea’in my_set
True
>>>‘river’not in my_set
False
>>>
```

Using Built-in Functions with Set

There are several Python functions that are often used with set to carry out various tasks.

Len()

Returns the number of elements on a set.

```
>>>my_set = {1, 'a', 2, 'b', 3, 'c'}
>>>len(my_set)
6
>>>
```

Max()

Returns the largest element on a set.

```
>>>my_set = {1,2,3,4,5}
>>>max(my_set)
5
>>>
```

On a set of strings, max() returns the last item alphabetical-wise.

```
>>> b = {'a', 'b', 'c', 'd', 'e'}
>>>max(b)
'e'
>>>
```

Min()

Returns the smallest element on a set.

```
>>> a = {2, 1, 5, 8, 9, 20}
>>>min(a)
1
>>>
```

```
>>> b = {'a', 'b', 'c', 'd', 'e'}
>>>min(b)
'a'
>>>
```

Sorted()

Returns a sorted list of set elements but does not actually sort the set.

```
>>> my_set = {‘red’, ‘blue’, ‘yellow’, ‘green’, ‘violet’}
>>>sorted(my_set)
[‘blue’, ‘green’, ‘red’, ‘violet’, ‘yellow’]
>>>
```

Using sorted() does not affect the actual order of the elements on the set:

```
>>> my_set
{‘violet’, ‘green’, ‘yellow’, ‘red’, ‘blue’}
>>>
```

Sum()

Returns the total of all items on a set.

```
>>> a = {3, 1, 7, 2, 4, 10}
>>>sum(a)
27
>>>
```

Enumerate()

Returns an enumerate object which contains the value and index of all set elements as a pair.

```
>>> my_set = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’}
>>>enumerate(my_set)
<enumerate object at 0x02D9A760>
>>>
```

Iterating Through Sets

You can iterate through every element on a set with a ‘for loop’.

```
for letter in set(‘programmer’):
        print(letter)
```

g

m

r

p

a

e

o

Frozenset

A frozenset takes the characteristics of a set but has immutable elements. Once assigned, you can no longer change its elements. A frozenset relates to a set as a tuple relates to a list. Frozensets, unlike sets, are hashable and can thus be used as dictionary keys. Their immutable nature does not allow the use of Python methods that add or remove items within the frozenset.

```
>>> x = frozenset ([9, 7, 5, 3, 1])
>>> y = frozenset ([6, 5, 1, 10, 2])
>>> x.difference(y)
frozenset({9, 3, 7})
>>> x | y
frozenset({1, 2, 3, 5, 6, 7, 9, 10})
>>> x.isdisjoint(y)
False
>>>
```

Step 12: Dictionary

A dictionary is **an unordered collection of key-value pairs** which are **separated by a colon** and **enclosed within curly braces {}**. A dictionary may contain any data type and is mutable. Its keys, however, are immutable and can only be a string, tuple, or a number. Values stored on a dictionary can only be accessed through the keys.

A dictionary is used as a container for storing, managing, and retrieving data in key-value format found in phone books, menus, or directories. Python provides a number of operators that can be used to perform different tasks with a dictionary.

A basic dictionary has this structure:

d = {key_1:y, key_2:1.5, key_3:xyz, key_4:7.55}

You can create an empty dictionary with this syntax:

d = {}

Accessing Elements on a Dictionary

Since a dictionary is an unordered data type, you can’t use indexing to access values. You will instead use its keys. To access data, you can either place the **keys[]**inside square brackets or use the **get() method**.

To access date by using keys:

```
>>> my_dict = {‘Name’:‘Mark’, ‘Age’: 24, ‘Ranking’: ‘5th’, ‘Average’:89.5}
>>>my_dict[‘Name’]
‘Mark’
>>> my_dict[‘Ranking’]
‘5th’
>>>
```

Here we have accessed the data for they keys Name and Ranking.

In this example;

- **my_dict** is the dictionary
- Name, Age, Ranking and Average are **keys**
- Mark, 24, 5th and 89.5 are **values (data)**
- There are **4 key-value pairs**:

- Name:Mark
- Age:24
- Ranking:5th
- Average:89.5

To access the same values with the **get() method**:

```
>>> my_dict = {'Name':'Mark', 'Age': 24, 'Ranking': '5th', 'Average':89.5}
>>> my_dict.get('Name')
'Mark'
>>> my_dict.get('Average')
89.5
>>>
```

Adding and Modifying Entries to a Dictionary

You can easily add new items or modify the value of existing keys with the assignment **operator =**. When you assign a key:value pair to a dictionary, Python checks whether the key already exists on the dictionary or not. If there is a similar key, the value simply gets updated. If it's a unique key, then the key:value pair is added to the dictionary.

The **syntax** for adding a new entry to a dictionary is:

dict_name[key] = b

For example, using the same dictionary above, you can **add a new key-value pair**, status: regular:

```
>>> my_dict = {'Name':'Mark', 'Age': 24, 'Ranking': '5th', 'Average':89.5}
>>> my_dict['status'] = 'regular'
>>>
```

To check the updated dictionary:

```
>>> my_dict
{'Name': 'Mark', 'status': 'regular', 'Ranking': '5th', 'Age': 24, 'Average': 89.5}
>>>
```

To **modify a value** stored in a dictionary key, you can assign a new value to the key using the assignment operator. For example, you can change the value of the Ranking from 5th to 3rd with:

```
>>> my_dict = {'Name':'Mark', 'Age': 24, 'Ranking': '5th', 'Average':89.5}
>>> my_dict['Ranking'] = '3rd'
>>> my_dict
>>>{'Ranking': '3rd', 'Name': 'Mark', 'Average': 89.5, 'Age': 24}
>>>
```

Removing or Deleting Elements from a Dictionary

To remove a key: value pair from a dictionary, you can use the **pop() method** which removes the pair and returns the value of the given key.

The pop()method

For example:

```
>>>my_dict = {'ocean': 'Pacific Ocean', 'Sea': 'Baltic Sea', 'river':'Danube', 'swamp': 'The Everglades'}
>>>
```

To remove the key 'river' and its value:

```
>>>my_dict.pop('river')
'Danube'
>>>
```

Here's the updated my_dict:

```
>>> my_dict
{‘Sea’: ‘Baltic Sea’, ‘swamp’: ‘The Everglades’, ‘ocean’: ‘Pacific Ocean’}
>>>
```

The popitem() method

The **popitem() method** is another way of removing key: value pairs on a dictionary. This method takes no argument and removes and returns an arbitrary pair from the dictionary.

For example, when you use dict.popitem() on my_dict, you’ll have no control over the value it will remove:

```
>>>my_dict = {‘ocean’: ‘Pacific Ocean’, ‘Sea’: ‘Baltic Sea’, ‘river’:‘Danube’, ‘swamp’: ‘The Everglades’}
>>> my_dict.popitem()
(‘swamp’, ‘The Everglades’)
>>>
```

Here’s what’s left of my_dict:

```
>>> my_dict
{‘ocean’: ‘Pacific Ocean’, ‘river’: ‘Danube’, ‘Sea’: ‘Baltic Sea’}
>>>
```

The clear() method

Only two key: value pairs remain in my_dict. To remove the remaining pairs at once, you can use the **clear() method**:

```
>>> my_dict.clear()
>>>
```

You now have an empty dictionary:

```
>>> my_dict
{}
>>>
```

To delete the my_dict dictionary, you will use the **del keyword**:

```
>>>del my_dict
>>>
```

If you try to access my_dict again, Python will raise a NameError:

```
>>> my_dict
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    my_dict
NameError: name 'my_dict' is not defined
>>>
```

Other Python Dictionary Methods

Earlier, you have learned methods that can be used to add, change, remove, or clear keys and values on a dictionary. There are other methods that you can use to perform various tasks in Python.

Update(other)

Updates a dictionary with key-value pairs from another dictionary.

The update(other) method updates a dictionary with a set of key-value pairs from another dictionary. It merges the key value pairs of one dictionary into another and allows the values of the other dictionary to overwrite the values of the current dictionary in situations where a common key() exists.

For example:

```
>>> dict_1 = {'First Name':'Chuck', 'Age': 27, 'Branch':'Chicago'}
>>>dict_2 = {'Last Name': 'Davidson', 'Position': 'Supervisor', 'Branch':'New York'}
>>> dict_1.update(dict_2)
```

The key:value pairs of dict_2 have now been merged with dict_1:

>>> dict_1

{'Age': 27, 'Position': 'Supervisor', 'First Name': 'Chuck', 'Last Name': 'Davidson', 'Branch': 'New York'}

>>>

Take note that there is one common key between dict_1 and dict_2, the Branch. Hence, the value of dict_2, New York, replaced the original value on dict_1, Chicago.

Since only the dict_1 dictionary was updated, the key-value pairs of dict_2 remains the same:

>>> dict_2

{'Position': 'Supervisor', 'Branch': 'New York', 'Last Name': 'Davidson'}

>>>

Item() method

Returns list of a dictionary's key-value pairs.

The **syntax** is:

```
dict.items()
```

>>> x = {1:"abc", 2:"def", 3:"ghi", 4:"jkl"}

>>> x.items()

dict_items([(1, 'abc'), (2, 'def'), (3, 'ghi'), (4, 'jkl')])

>>>

Values() method

Returns a list of dictionary values.

The **syntax** is:

```
dict.values()
```

```
>>> y = {1:"xyz",2:"aeiou", 3:"uvw", 4:"rst"}
>>> y.values()
dict_values(['xyz', 'aeiou', 'uvw', 'rst'])
>>>
```

Keys() method

Returns a list of dictionary keys.

The **syntax** is:

```
dict.keys()
```

```
>>> dict_one = {'animal': 'tiger', 'age':4, 'location':'Cage 5'}
>>> dict_one.keys()
dict_keys(['animal', 'location', 'age'])
>>>
```

Setdefault() method

Searches for a given key in a dictionary and returns the value if found.

If not, it returns the given default value.

The **syntax** for this method is:

```
dict.setdefault(key, default=None)
```

```
>>> my_dict = {'a':'car', 'b':'van', 'c':'yacth', 'd':'bus'}
>>> my_dict.setdefault('c', None)
'yacth'
>>> my_dict.setdefault('f', None)
>>>
```

Copy() method

Returns a shallow copy of a dictionary.

The copy() method performs a shallow copy of a dictionary where every key-value pair is duplicated. The method allows users to modify the dictionary copy without altering the original file.

To illustrate, here is a series of statements showing how the copy method is used to produce a shallow copy of the original dictionary.

Here is the original dictionary:

```
>>>my_dict1 = {“apples”: 10, “oranges”: 5, “grapefruit”: 7, “strawberry”: 12}
>>>
```

Now you can create a copy of my_dict:

```
>>>my_dict2 = my_dict1.copy()
>>>
```

A dictionary copy is a new file which is independent of the original dictionary from which it was generated. Hence, any changes you make to the new dictionary will have no effect at all on the original dictionary.

Now, modify the dictionary copy:

```
>>>my_dict2[“oranges”] = 50
>>>my_dict2[“apples”] = 5
>>>
```

Now, print my_dict1 and my_dict2:

```
>>>print(my_dict1)
>>>print(my_dict2)
>>>
```

Your screen will display these details:

{‘oranges’: 5, ‘grapefruit’: 7, ‘apples’: 10, ‘strawberry’: 12}

{‘oranges’: 50, ‘apples’: 5, ‘grapefruit’: 7, ‘strawberry’: 12}

The fromkeys() method

Takes items on a sequence and uses them as keys to build a new dictionary.

The fromkeys() method takes a sequence of items and uses them as keys to create a new dictionary. It allows a second argument through which you can provide a value that will be attached to the keys on the new dictionary.

Here is the list that will be used as keys:

```
>>>keys = [“monitor”, “CPU”, “mouse”, “keyboard”,  “speaker”]
>>>
```

Now, create a new dictionary from keys:

```
>>>new_dict = dict.fromkeys(keys, 10)
>>>
```

Display the key-value pairs of the new dictionary:

```
>>>print(new_dict)
```

This will be the output on your screen:

```
{‘keyboard’: 10, ‘speaker’: 10, ‘monitor’: 10, ‘CPU’: 10, ‘mouse’: 10}
>>>
```

Dictionary Membership Test

You can use the membership operators ‘in’ and ‘not in’ to check whether a specific key exists or not on the dictionary. Take note that this test is only for dictionary keys, not values.

Examples:

```
>>>even = {2:‘GRQ‘, 4:‘XYZ’, 6:‘DEF’, 8:‘GHI’, 10:‘JKL’}
>>>2 in even
True
>>> 12 in even
False
>>> 8 not in even
False
>>> 14 not in even
True
>>>
```

Iterating Through a Dictionary

You can use the ‘for’ lop to iterate through a dictionary. For example:

```
for n in numbers:
        print(numbers[n])
```

```
2
10
8
6
4
```

Using Built-in Functions with Dictionary

Python has several built-in functions that can be used with dictionary to perform various tasks.

Lens()

Returns the number of items on a dictionary.

```
>>>dict_one = {“a”:“Requirements”, “b”:“Name”, “c”:“Age”, “d”:“Grade”}
>>>len(dict_one)
```

4

>>>

Sorted()

Returns a sorted view of dictionary keys but does not sort the dictionary itself.

```
>>> my_dict = {"color":"silver", "year model": 2012, "warehouse":"used"}
>>>sorted(my_dict)
['color', 'warehouse', 'year model']
>>> my_dict
{'color': 'silver', 'year model': 2012, 'warehouse': 'used'}
>>>
```

Creating a Dictionary with the dict() function

Another way of creating a dictionary is with **dict()**, a built-in function. The function allows programmers to create a dictionary out of a list of tuple pairs. Each pair will have two elements that can be used as a key and a value.

First, create a list of tuple pairs that are key-value pairs.

```
>>>pairs = [("cat", "kitten"), ("dog", "puppy"), ("lamb", "ewe"), ("lion", "cub")]
>>>
```

Then, convert a list to a dictionary.

```
>>>dict(pairs)
{'dog': 'puppy', 'lion': 'cub', 'cat': 'kitten', 'lamb': 'ewe'}
>>>
```

Dictionary Comprehension

Dictionary comprehension is a concise way of creating a new dictionary from a Python iterable. It consists of a key-value expression and a 'for statement' enclosed in curly braces {}. Following is an example that shows how you can build a dictionary with key-value pairs of a range of numbers and their square value:

```
>>> squares = {x: x**2 for x in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

```
>>>
```

A dictionary comprehension mayhold more than one conditional statement (for or if statements). In the above statement, an ‘if statement’ may be added to filter out desired items to build a new dictionary. Following are some examples:

Example #1:

```
>>> even_squares = {x: x**2 for x in range(12) if x%2 == 0}
>>> even_squares
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
>>>
```

Example #2:

```
>>>odd_squares = {x: x**2 for x in range(13) if x%2 == 1}
>>> odd_squares
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81, 11: 121}
>>>
```

Step 13: Python Operators

Operators are special symbols that indicate the implementation of a specific process. They are used to evaluate, manipulate, assign, or perform mathematical or logical operations on different data types. Python supports different types of operators:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Identity Operators
- Membership Operators
- Bitwise Operators

Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations on numeric data types:

+	Addition	Adds the value of operands on either side of the operator . Example: 2 + 8 returns 10
-	Subtraction	Subtracts the right operand from the left operand. Example: 10 – 2 returns 2
*	Multiplication	Multiplies the value of the left and right operands. Example: 3 * 4 returns 12
/	Division	Divides the left operand with the value of the right operand. Example: 20 / 5 returns 4

**	Exponent	Performs exponential calculation. Example: 2 ** 3 (two raised to the power of 3) returns 8
%	Modulos	Returns the remainder after dividing the left operand with the value of the right operand. Example: 13 % 3 returns 1
//	Floor Division	Divides the left operand with the right operand and returns a quotient stripped of decimal numbers. Example: 13 // 3 returns 4

Assignment Operators

Assignment operators are used to assign values to variables.

Operators	Function
=	Assigns the value of the right operand to the left operand. Examples: a = “xyz” x = 130 y = [2, 3, 5, 7, 8]
+= add and	Adds the left and right operands and assigns the value to the left operand, works like x = x + a Examples: x += 8, adder += 4
-= subtract and	Subtracts the right operand from the left operand and assigns the difference to the left operand, works like x = x -8 Examples: x -= 4, counter -= 6
*= multiply and	multiplies the left and right operands and assigns the product to the left operand, works like x = x * 2 Examples: x *= 4, product *= 5
/= divide and	Divides the left operand with the value of the right operand and assigns the result to the left operand, works like x = x / 4 Examples: x /= 4, counter /= 2
**= exponent	Performs exponential operation on the left operand and assigns the value to the left operand, works like x = x**2 Examples: x **= 3, double **= 2
	Performs floor division on the left operand and assigns the value to the left operand, works like x = x//2

//= floor division and	
	Examples: x //= 3, amount //= 2

Relational or Comparison Operators

Relational operators evaluate the given comparison expression and return the relation of left and right operands as either True or False.

Python supports the following relational or comparison operators:

Operators	**Meaning**
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Logical Operators

There are three types of logical operators in Python:

or

and

not

Python evaluates expressions with logical operators in this manner:

x or y	If the first argument (x) is true, it returns True. If x is false, it evaluates the second argument, y, and returns the result.
x and y	If x is true, it evaluates y. If y is false, it returns False. If x is false, it returns False.
not x	It returns True if x is false and False if x is true.

Examples:

```
>>>(12>9) or (2<8)
True
>>> (12 > 28) or (5 < 7)
True
>>> (8>2) and (6>15)
False
>>> (6 ==2* 3) and ( 10 < 5 **3)
True
>>>not (3 * 5 > 4)
False
>>>not (5> 2**4)
True
>>>
```

Identity Operators

Identity operators are used to verify if two objects are stored in the same memory location. Python has two identity operators:

Operators	Description
is	Returns True if the variables on the left and right side of the operator refer to the same object; returns False if otherwise.
is not	Returns False if the variables on the left and right side of the operator refer to the same object; returns True if otherwise.

To illustrate how identity operators work, type the following expressions on the text editor then save and run the program:

```
>>>x = 10
>>>y = 10
>>>a = “Programs”
>>>b = “Programs”
>>>x1 = [4, 6, 8]
>>>y1 = [4, 6, 8]
>>>z1 = [4, 5, 8]
>>>print(x is y)
>>>print(a is not b)
>>>print(x1 is y1)
>>>print(x1 is not y1)
>>>print(x1 is z1)
>>>
```

You screen would display the following results:

```
True
False
```

False

True

False

Variables x and y hold integers of the same value. They are, therefore, both identical and equal. Hence, the ‘is’ id operator returned True.

Variables a and b contains the same string and data type and are identical. Hence, the ‘is not’ operator returned False.

The variables x1 and y1 refer to a list. Although they have the same value, they are not identical because the lists are mutable and are stored in different memory locations. Hence, the use of the id operator ‘is’ returned False. Conversely, the use of the ‘is not’ operator on x1 and y1 returned True. Using the ‘is’ operator on variables x1 and z1 would naturally result to False because not only are they mutable data types; they actually hold different items.

Membership Operators

Python’s membership operators are used to test for the occurrence or non-occurrence of a variable or value in a sequence which can be a string, tuple, list, or a dictionary. In the case of a dictionary, you can only test for the occurrence of a key but not its value. There are two types of membership operators in Python:

Operators	Description
in	Returns True if the specified variable or value is found on a given sequence; returns False if otherwise
not in	Returns True if the specified variable or value is not found on a given sequence; returns False if otherwise.

To illustrate, type the following on the text editor and run the code:

```
>>>my_string = ‘Membership operators can be used with a string.’
>>>my_dict = {“animal” : ‘elephant’, “size” : ‘large’, “color” : ‘gray’ }
>>>print(‘p’in my_string)
```

```
>>>print('membership'in my_string)
>>>print('x' not in my_string)
>>>print('color'in my_dict)
>>>print("animal" not in my_dict)
>>>print('elephant'in my_dict)
>>>
```

You will get these results:

```
True
False
True
True
False
False
```

There is a 'p' in my_string, so the interpreter returned True. In fact, there are two p's. In the second print statement, the interpreter returned false because there is no 'membership' substring in my_string. What you have is 'Membership'. Remember that Python is a case-sensitive language. There is a 'color' key in my_dict so Python returned True. There is a key named 'animal' so it returned False when the 'is not' operator was used. On the last print statement, 'elephant' is a value, not a key on the my_dict dictionary. Hence, the interpreter returned False.

Bitwise Operators

In computers, a series of zeros and ones called bits represent numbers. Bitwise operators work on operands like they are strings of binary digits. They are used to directly manipulate bits.

Understanding the Base 2 Number System

In normal life, counting is done in base 1o. This means that for every number, each place can contain one of ten values from zero to nine and that you carry over to the next place when it goes higher than nine. In binary, counting is done in base two where a number place can hold either zero or one. Like the counting pattern in base 1o, you carry over to the next place every time the count goes over one.

For example, the numbers 0 and 1 are represented similarly in base 10 and base 2. In base 2, however, you have to carry over and add 1 to the next number place when the count exceeds 1. Hence, in base2, number 2 is represented as '10' and 3 is '11'. When you go on

to 4, you have to carry over to the next number place. Hence, 4 is represented as ‘1o0’ in base 2.

In base 10, a decimal place denotes a power of ten. In base 2, each place represents a bit or a power of two. The 1’s bit (rightmost bit) denotes ‘two to the zero power’ while the 2’s bit (next bit) denotes ‘two to the first power’. The succeeding bits are 4, 8, 16, and so on.

In Python, numbers are written in binary format with the prefix 0b. To print an integer in its binary format, you can use the bin() function which takes integer as argument and returns it as a string. Once you use the bin() on a number, you can no longer work on it as a number.

Here are the bitwise operators in Python:

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Bitwise left shift
>>	Bitwise right shift

Examples:

Assuming x = 12 (1100 in binary) and y = 5 (101 in binary):

& Bitwise AND

```
>>> x = 12
```

```
>>>y = 5
>>> x & y
4
>>>bin(4)
'0b100'
>>>
```

Take note that whether you use the integer or the binary format (with 0b prefix) in values that you assign to the variables x and y, Python will return an integer when you use the bitwise operators on the variables.

| Bitwise OR

```
>>> x = 12
>>>y = 5
>>>x | y
13
>>>bin(13)
'0b1101'
>>>
```

^ Bitwise XOR

```
>>> x = 12
>>>y = 5
>>> x ^ y
9
>>>bin(9)
'0b1001'
>>>
```

~ Bitwise NOT

```
>>> x = 12
>>>y = 5
>>> ~ x
```

```
-13
>>>bin(-13)
‘-0b1101’
>>> ~ y
-6
>>>bin(-6)
‘-0b110’
>>>
```

<<   Bitwise left shift

```
>>> x = 12
>>>y = 5
>>> x << 2
48
>>>bin(48)
‘0b110000’
>>> y << 2
20
>>>bin(20)
‘0b10100’
>>>
```

>> Bitwise right shift

```
>>> x = 12
>>>y = 5
>>> x >> 2
3
>>>bin(3)
‘0b11’
>>>y >> 2
1
```

```
>>>bin(1)
'0b1'
>>>
```

Precedence of Operators

Operator precedence impacts how Python evaluates expressions. For example, in evaluating the expression x = 20 – 5 * 3, the value of 5 instead of 45 will be stored to the variable x. This is because the multiplication operation 5 * 3 has precedence over the subtraction operation 20 - 5. In the following table, operators are arranged in the order of precedence from the highest to the lowest:

Description	**Operators**
Exponentiation	**
Complement, unary plus, and minus	~ + -
Multiplication, division, modulo, and floor division	* / % //
addition and subtraction	+ -
Right and left bitwise shift	>>, <<
Bitwise ‘AND’	&
Regular `OR’ and Bitwise exclusive ‘OR’	| ^
Comparison operators	>, <, >=, <=
Equality operators	== , !=
Assignment operators	=, +=, -=, *-, /=, %= //= **=

Identity Operators	is, is not
Membership operators	in, not in
Logical operators	OR, AND, NOT

Step 14: Built-in Functions

Python comes with several built-in functions that you can readily use to create useful programs. You have learned some of them in earlier lessons. In this section, you will learn the most commonly used functions.

Here is a tabulation of all built-in functions in Python 3:

abs()	all()	ascii()	any()
bin()	bool()	bytes()	bytearray()
callable()	chr()	compile()	classmethod()
complex()	delattr()	dir()	dict()
divmod()	enumerate()	exec()	eval()
filter()	format()	float()	frozenset()
global()	getattr()	hasattr()	hash()
hex()	help()	__import__()	id()
input()	int()	issubclass()	isinstance()
iter()	list()	len()	locals()
max()	map()	min()	memoryview()

next()	object()	open()	oct()
ord()	print()	pow()	property()
repr()	range()	round()	reversed()
set()	slice()	setattr()	sorted()
str()	sum()	staticmethod()	super()
type()	tuple()	vars()	zip()

The range() function

The range() function is used to create lists that contain arithmetic progressions. This versatile function is most frequently used in for loops. The syntax is range(start, end, step) where all arguments are plain integers. When only one argument is given, Python ascribes it as the 'end' argument. Omission of the start argument sets the start at the default value of zero. Omission of the step argument sets progression to the default value of 1.

The range expression generates an iterator which progresses a set of integers from a given or default starting value to an ending value.

To illustrate how this function works, type range(10) on the command line:

```
>>>range(10)
range(0, 10)
>>>
```

Since both the start and progression values were omitted, the list of numbers starts at the default value of zero and the step is set to the default value of 1.

To see the list of numbers on the given range, you will use the expression list(range(n)).

```
>>>list (range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Notice that the range displayed ended in an integer (9) which is one less than the ending argument (10) of the range expression.

Here is a range expression with 3 arguments:

```
>>>range(2, 34, 2)
range(2, 34, 2)
>>>
```

```
>>>list (range(2, 34, 2))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32]
>>>
```

Take note that the list generated ended at an integer (32) which is two (the step argument) less than the ending argument (34).

Other examples:

```
>>>range(0, -14, -1)
range(0, -14, -1)
>>> list (range(0, -14, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13]
>>> list(range(1,0))
[]
>>>list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>>
```

The input() Function

Most programs require the user's input to work. Input can come from a variety of sources such as keyboard, mouse clicks, the internet, database, or external storage with the

keyboard as the most commonly used channel. Python's input() function handles users' response through the keyboard.

The input() function has an optional parameter, a prompt string. Once the function is called, the prompt string is displayed onscreen as the program awaits user's input. Python returns the user's response as a string.

To illustrate, here is a program snippet that collects keyboard input for first name, last name, and occupation:

```
name1 = input('Please enter your first name: ')
name2 = input('Please enter your last name: ')
print('Good day, ' + name1 + ' ' + name2 + '!')
occupation = input('What is your occupation? ')
print('So you are a/an ' + occupation + ', ' + name1 + '.' + ' Great!')
```

When you run the program, the first prompt string will be displayed onscreen as:

Please enter your first name:

At this point, the program is waiting for the user's response and does nothing until a keyboard input is received. Type 'Kurt'. The program will return the response immediately after the prompt string:

Please enter your first name: Kurt

After displaying 'Kurt', the user's response, the program now proceeds to display the next prompt string:

Please enter your last name:

Type the name 'Johnson' as your response to the prompt.

Immediately after returning'Johnson', the program will execute the next line, a print statement. It will then display the next prompt string.

Good day, Kurt Johnson!

What is your occupation?

Type the word student in response to the prompt, as in the following:

What is your occupation? student

After returning ‘student’, the program will process the next line, another print statement:

So you are a/an student, Kurt. Great!

After the program’s execution, this is what you will see on your monitor:

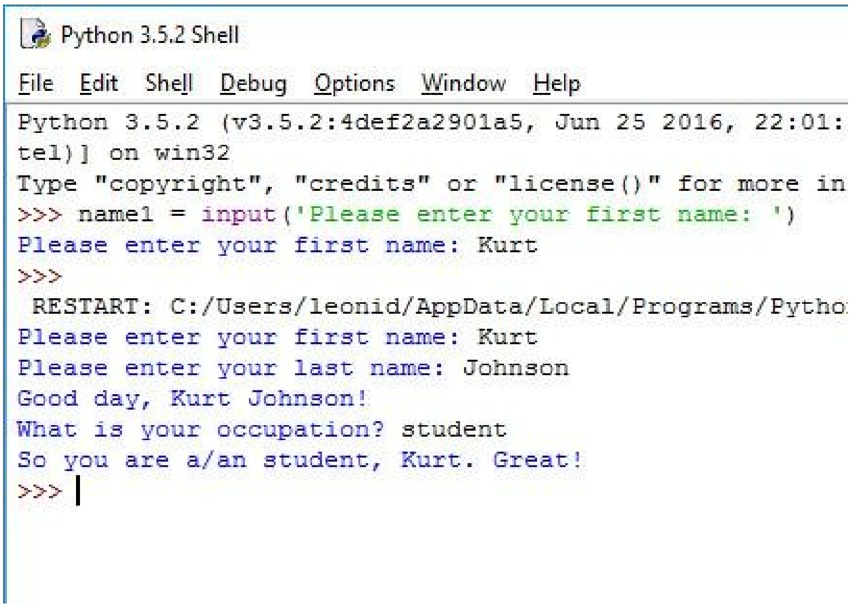
Please enter your first name: Kurt

Please enter your last name: Johnson

Good day, Kurt Johnson!

What is your occupation? student

So you are a/an student, Kurt. Great!



When writing your own program, make sure that you provide white spaces in appropriate places. This is important for better readability. You wouldn’t want the user’s input to crowd the prompt string so you have to provide a space before the closing quotation mark. For instance, if you fail to allocate white spaces in the above example, the whole interaction will look like this:

Please enter your first name:Kurt

Please enter your last name:Johnson

Good day,KurtJohnson!

What is your occupation?student

So you are a/anstudent,Kurt.Great!

Password Verification Program

This simple program will put together what you have learned about input(), dictionary, and print().

Likewise, it will introduce you to an if…else code block, one of the conditional structures supported by Python.

An if…else block first evaluates the expression given in the 'if' statement. If the test condition is True, the statement(s) in the 'if' body is executed. If False, the statement(s) in the 'else' block is executed.

This program makes use of a dictionary to store key:value pairs of usernames and password. It uses the input() function to obtain the username then applies the membership test to check if the response matches with one of the keys in the dictionary. If yes, then the program uses the get() method to retrieve the value stored in the given key and stores the value in the variable pword. It then prompts the user to enter a password and stores the response in the variable password. Next, the program tests if pword is equal to pword and prints a 'Thank you!' string if equal. If not, it prints a string that asks the user to try again.

The 'else' statement at the end of the program is executed if the given username does not match any of the keys in the dictionary.

```
usernames = {‘Adrian’:‘123456’, ‘John’:‘GVEST’, ‘Richard’:‘REJ321’, ‘Caleb’:‘875’}

user = input(“Please enter your username: “)
if user in usernames:
```

```
    pword = usernames.get(user)

    password = input("Please enter your password: ")

    if password == pword:

        print('Thank you!')

    else:

        print('You entered an incorrect password. Please try again.')



else:

    print('You are not a registered user.')
```

If you run the program, you may see the following results:

```
Please enter your username: Adrian
Please enter your password: 123456
Thank you!
>>>

Please enter your username: Caleb
Please enter your password: QED
You entered an incorrect password. Please try again.
>>>
```

Please enter your username: Michelle

You are not a registered user.

>>>

Using input() to add elements to a List

This program uses input to append the user's response to an existing list then prints the updated number of members and the items stored on the updated list.

```
members = [“Marc”, “Jane”, “Joshua”, “Kian”, “May”, “Jessica”]
name = input(“Please enter your name: “)
print(“Thanks for joining the Student Organization, ” + name + “!”)
members.append(name)
total = len(members)
totalstr = str(total)
print(“There are now ” + totalstr + ” members: “)
print(members)
```

Run the program. At the prompt for name, enter Jasmine.

Here is what Python returns:

Please enter your name: Jasmine

Thanks for joining the Student Organization, Jasmine!

There are now 7 members:

[‘Marc’, ‘Jane’, ‘Joshua’, ‘Kian’, ‘May’, ‘Jessica’, ‘Jasmine’]

>>>

A program that sorts on ascending basis words from a string entered by a user:

First, ask the user to enter a string:

```
str = input(“Enter a sentence: “)
```

Then split the string to form a list of words:

```
words = str.split()
```

Then sort the list:

```
words.sort()
```

Then print the sorted list:

```
for x in words:
  print(x)
```

Run the program and enter a sentence after the string prompt.

Enter a sentence: Python has many useful string methods.

After evaluating the string entered, Python displays each word in the sentence as a list of words:

Python

has

many

methods.

string

useful

>>>

The print() Function

For the interpreter to recognize the print function, you have to enclose the print parameters inside parentheses.

Examples:

```
>>> print("This is the print function.")
This is the print function.
>>> print(15)
15
>>>print(3**3)
27
>>>
```

Python can print multiple values within the parentheses. The values must be separated by a comma.

To illustrate, here are three variable assignment statements:

```
>>>a = "employee"
>>>b = "age"
>>>c = 25
>>>print("requirements : ", a, b, c)
```

Here's the output:

```
requirements :  employee age 25
```

abs()

The abs() function returns the absolute value of integers or floats. The value returned is always a positive number.

Examples:

```
>>>abs(-18)
18
```

```
>>> abs(30)
30
>>>abs(-88.5)
88.5
>>>
```

The abs() function returns the magnitude when the arguments used are complex numbers.

```
>>>abs(2 + 2j)
2.8284271247461903
>>>abs(1 - 3j)
3.1622776601683795
>>>
```

max()

The max() function returns the largest value among two or more numeric type data given as arguments.

Examples:

```
>>>max(-100, 4, 25, 17)
25
>>>max(2o, -50, 5, -70)
20
>>>
```

min()

The min() function returns the least value among two or more numeric data types given as arguments.

Examples:

```
>>>min(10, -60, 5, 0)
-60
>>>min(4, 0, 65, 1)
```

```
0
>>>
```

type()

The type() function returns the data type of the given argument.

Examples:

```
>>>type(“Python is a powerful programming language.”)
<class ‘str’>
>>>type(25)
<class ‘int’>
>>>type(40.5)
<class ‘float’>
>>>type(2 +3j)
<class ‘complex’>
>>>
```

Step 15: Conditional Statements

Decision making structures are necessary in situations when you want your program to perform an action or a calculation only when a certain condition is met. Decision making constructs begin with a Boolean expression, an expression that returns either True or False. The response will be used as a basis for determining how the program flows. Python supports the following conditional statements:

if statements

if else statements

elif statements

else

nested if…elif…else statements

if statements

An if statement stars with a Boolean expression followed by a statement or a group of statements that tells what action should be done if the test expression is True. An if statement uses the following **syntax**:

```
if expression:
    statement(s)
```

Python evaluates the 'if' expression and executes the body of the program only if the evaluation is True. You must take note of the indentation of the statements on the body of the if expression. The first unindented line indicates the end of an if block.

To illustrate, here's a program that collects keyboard input and uses the response as the basis for succeeding actions.

This program asks the user to enter a vowel and prints a string if the input meets the condition.

```
vowels = [‘A’, ‘E’, ‘I’, ‘O’, ‘U’, ‘a’, ‘e’,‘i’, ‘o’, ‘u’]

letter = input(“Enter a vowel: “)

if letter in vowels:

    print(“Thank you, you may use this computer.”)

print(“You may only enter one of the vowels to proceed.”)
```

Here’s the output when you enter a vowel in response to the prompt:

```
Enter a vowel: o
Thank you, you may use this computer.
You may only enter one of the vowels to proceed.
>>>
```

Assuming you enter the consonant H, here’s what the output would be:

```
Enter a vowel: H
You may only enter one of the vowels to proceed.
```

if…else statements

An if…else statement block first evaluates the ‘if expression’. If the test condition is True, Python executes the statements in the body of the ‘if statement’. Otherwise, if the condition is False, Python executes the statements in the else block.

An if…else statement has the following **syntax**:

```
if test expression:
```

```
    statement(s)
else:
    statement(s)
```

To illustrate, here's a program which uses the 'if..else' structure:

This program checks if a food order is on stock and prints appropriate string

```
stock = [hamburger', 'pizza', 'hotdog', 'barbeque']

order = input("Please enter your order: ")
if order in stock:
    print("Thank you. Your order will be served in 5 minutes.")
else:
    print("Sorry, we don't serve " + order + " at the moment."
```

If you run the program and enter pizza at the prompt, here's what it does:

Please enter your order: pizza
Thank you. Your order will be served in 5 minutes.

Run the program again. This time, enter something that's not on stock, 'ice cream'.

Here's the output:

Please enter your order: ice cream

Sorry, we don't serve ice cream at the moment.

if…elif…else statements

An elif (else if) statement can be used when there is a need to check or evaluate multiple expressions. An if…elif…else structure first checks if the 'if statement' is True.

If true, then Python executes the statements in the if block. If False, it tests the condition in the elif block. If the elif statement is evaluated as True, Python executes the statements in the elif block. Otherwise, control passes to the else block. An if block can have as many elif blocks as needed but it can only have one else block.

An if…elif…else statement has the following **syntax:**

```
if expression:
    if block
elif expression:
    elif block
else:
    else block
```

To illustrate, here is a simple program with an if…elif..else structure:

This program checks if a food order is in the list of foods that can be served in 5

Minutes. If not, it checks if the food order is in the list of foods that can be served in 15 minutes. If the food is not found in either list of foods, program prints an appropriate statement.

```
stock1 = [‘hamburger’, ‘pizza’, ‘hotdog’, ‘barbeque’]
stock2 = [‘Fried Chicken’, ‘French Fries’, ‘Chips’, ‘Apple Pie’]

order = input(“Please enter your order: “)
if order in stock1:
    print(“Thank you. Your order will be served in 5 minutes.”)
elif order in stock2:
    print(“Are you willing to wait? Your order will be served in 15 minutes.”)
else:
    print(“Sorry, we don’t serve ” + order + ” at the moment.”)
```

If you run the program and enter ‘hamburger’, here’s what the program does:

```
Please enter your order: hamburger
Thank you. Your order will be served in 5 minutes.
```

Run the program again and enter ‘Fried Chicken’, an item in the list stock2:

Please enter your order: Fried Chicken

Are you willing to wait? Your order will be served in 15 minutes.

If you run the program again and enter a food item that’s not on either list, Mango Pie, here’s what the output would be:

Please enter your order: Mango Pie

Sorry, we don’t serve Mango Pie at the moment.

nested if…elif…else statements

In programming, nesting is the practice of organizing information, sequence, loop, or logic structures in layers. Python allows conditional statements to contain another conditional statement. This structure is found in nested if…elif…else statements. Nested conditional statements are used whenever there is a need to check for another condition after the first condition has been evaluated as True. Nesting can go as deep as you would want it to go.

An if…elif…else statement has the following **syntax**:

```
if test_expression1:
   if test_expression1-a:
      statement_block1-a
   elif test_expression1-b:
      statement_block1-b
   else
```

```
    statement_block1-c
elif test_expression2:
  statement_block2
else:
  statement_block3
```

To illustrate, here is a program that uses the if…elif…else block:

This program asks for the user’s age and prints appropriate string based on the response.

```
num = int(input(“Enter your age: “))
if num >= 20:
   if num >= 60:
      print(“Please register with the Seniors Club.”)
   elif num >= 36:
      print(“You belong to the MiddleAgers Club.”)
   else:

```

```
        print(“Please register with the Young Adults Club.”)

elif num > 12:

    print(“You belong to the Youth Club.”)

else:

    print (“Sorry, you are too young to be a member.”)
```

Here’s how the program responds:

Enter your age: 65

Please register with the Seniors Club.

Enter your age: 36

You belong to the MiddleAgers Club.

Enter your age: 22

Please register with the Young Adults Club.

Enter your age: 13

You belong to the Youth Club.

Enter your age: 12

Sorry, you are too young to be a member.

Step 16:Python Loops

A loop is a control structure that allows the repetitive execution of a statement or group of statements. Loops facilitate complicated execution paths.

The for Loop

The ‘for loop’ is used to iterate over elements of sequential data types such as lists, strings, or tuples.

Its **syntax** is:

```
for val in sequence:
    statement(s)
```

In the for statement, the variable ‘val’stores the value of each item on the sequence with every iteration. The loop goes on until all elements are exhausted.

Examples:

For Loop with string:

```
>>>for letter in‘programming’:
print(‘<’, letter, ‘>’)
```

Here’s what you will see on your screen:

```
< p >
< r >
< o >
< g >
< r >
```

```
< a >
< m >
< m >
< i >
< n >
< g >
>>>
```

For Loop with list

```
weather = [‘sunny’, ‘windy’, ‘rainy’, ‘stormy’, ‘snowy’]

for item in weather:
    print(“It’s a”, item, “day!”)

print(“Dress appropriately!”)
```

This is the output when you run the loop:

```
It’s a sunny day!
It’s a windy day!
It’s a rainy day!
It’s a stormy day!
It’s a snowy day!
Dress appropriately!
>>>
```

for loop with a tuple

```
color = ('red', 'blue', 'pink', 'green', 'yellow')

for x in color:
    print("I'm wearing a", x, "shirt!")

print("Multi-colored shirts are cool!")
```

This is the output when you run the loop:

```
I'm wearing a red shirt!
I'm wearing a blue shirt!
I'm wearing a pink shirt!
I'm wearing a green shirt!
I'm wearing a yellow shirt!
Multi-colored shirts are cool!
>>>
```

Here is a loop that evaluates whether a number is even or odd. It prints the number and state if it is an even or odd number.

```
numbers = [10, 99, 3, 28, 41, 40, 5, 9, 66]

```

```
for n in numbers:
    if n % 2 == 0:
        print(n, “is an even number.”)
    else:
        print(n, “is an odd number.”)
```

```
10 is an even number.
99 is an odd number.
3 is an odd number.
28 is an even number.
41 is an odd number.
40 is an even number.
5 is an odd number.
9 is an odd number.
66 is an even number.
>>>
```

Using for loop with the range() function

The range() function can be used to provide the numbers required by a loop. For example, if you need the sum of 1 plus all the numbers form 1 up to 15:

```
x = 15
```

```
total = 0

for number in range(1, x+1):

    total += number


print(“Sum of 1 and numbers from 1 to %d: %d” % (x, total))
```

When you run the program, you will have this output:

```
Sum of 1 and numbers from 1 to 15: 120
>>>
```

The While Loop

The ‘while loop” is used when you need to repeatedly execute a statement or group of statements while the test condition is True. When the test condition is no longer true, program control passes to the line after the loop.

A while loop has this **syntax**:

```
while condition
    statement(s)
```

Here is a program that adds number up to num where num is entered by the user. The total = 1+2+3+4… up to the supplied number.

Example:

```
number = int(input("Enter a number: "))

#initialize total and counter
total = 0
c = 1

while c <= number:
    total = total + c
    c += 1

#print the total
print("The total is: ", total)
```

```
Enter a number: 5
The total is:  15
>>>
```

Break Statement

A break statement ends the present loop and instructs Python to execute the first statement

next to the loop. In nested loops, a break statement terminates the innermost loop and instructs the interpreter to execute the line that follows the terminated block. A break statement is commonly used to prevent the execution of the ‘else statement’. It is used to end the current iteration or the entire loop regardless of the test condition or when external conditions require immediate exit from the loop.

The **syntax** of the break statement is:

```
break
```

Here is a loop that ends once it reaches the word ‘sloth’:

```
animals = [‘lion’, ‘tiger’, ‘monkey’, ‘bear’, ‘sloth’, ‘elephant’]

for name in animals:
   if name == ‘sloth’:
      break
   print(‘Cool animal:’, name)
print(“Amazing animals!”)
```

Run the module to see this output:

Cool animal: lion

```
Cool animal: tiger
Cool animal: monkey
Cool animal: bear
Amazing animals!
>>>
```

Continue Statement

The continue statement skips remaining statement(s) in the present iteration and directs the program to the next iteration.

The **syntax** is:

```
continue
```

The break statement in the previous example may be replaced with the continue statement:

```
animals = [‘lion’, ‘tiger’, ‘monkey’, ‘bear’, ‘sloth’, ‘elephant’]

 for name in animals:
    if name == ‘sloth’:
       continue
    print(‘Cool animal:’, name)
print(“Amazing animals!”)
```

Here's the output:

```
Cool animal: lion
Cool animal: tiger
Cool animal: monkey
Cool animal: bear
Cool animal: elephant
Amazing animals!
>>>
```

Pass Statement

A pass is a null operation in Python. The interpreter reads and executes the pass statement but returns nothing. A pass statement is commonly used as a place holder whenever Python's syntax requires a line that you can't provide at the moment. It is used to mark codes that will eventually be written.

The **syntax** is:

```
pass
```

Examples:

Pass in an emptycode block:

```
for l in my_list:
    pass
```

Pass in an empty function block:

```
def my_function(a):
    pass
```

Pass as placeholder in an incomplete class block:

```
class Jobs:
    pass
```

Looping Techniques

The ‘for loop’ and the ‘while loop’ can be combined with loop control statements to create various loop forms.

Infinite loops (while loop)

The infinite loop is formed with the while statement. You’ll get an infinite loop when the specified test condition is always True. Following is an example of an infinite loop. The program imports the math module and uses the square root method on the given number. To leave the loop, you will have to press Ctrl-c.

IMPORTANT

We will discuss importing math modules in Step 18.

```
import math

while True:
    num = int(input(“Enter a number: “))
    print(“The square root of” ,num, “is”, math.sqrt(num))
```

Here’s the output:

Enter a number: 9

The square root of 9 is 3.0

Enter a number: 4

The square root of 4 is 2.0

```
Enter a number: 8
The square root of 8 is 2.8284271247461903
Enter a number: 49
The square root of 49 is 7.0
Enter a number:
>>>
```

Because the condition will always be True, the while loop will continue to execute and ask the user to enter a number.

Loops with top condition (while loop)

A while loop with a condition placed at the top is a standard while loop with no break statements. The loop ends when the condition becomes False. Here is a program that illustrates this form of while loop:

```
x = int(input(“Enter a number: “))

#initialize total and counter
total = 0
count = 1

while count <= x:
    total = total + count
    count = count + 1    # updates counter
```

```
# prints the total
print(“The total is”,total)
```

Here’s the output when you enter the numbers 4 and 7:

```
Enter a number: 4
The total is 10
>>>

Enter a number: 7
The total is 28
>>>
```

Loops with middle condition

This loop is usually implemented with an infinite loop and a conditional break in between the loop’s body.

This program takes input from user until the desired input is entered.

```
letters = “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz”

# infinite loop
while True:
    l = input(“Enter a letter: “)
```

```
    # condition in the middle

    if l in letters:

        break

    print("That is not a letter. Please try again!")


print("Perfect!")
```

Run the program and try entering a number. The program will run infinitely if you insist on entering a number.

Enter a letter: 5

That is not a letter. Please try again!

Enter a letter: 3

That is not a letter. Please try again!

>>>

Enter a letter and the break statement will be executed. Program flow will continue to the next line, a print statement.

Enter a letter: a

Perfect!

>>>

Loops with condition at the end

In this looping technique, the loop's body is run at least once. It can be run with an infinite loop and a conditional break at the bottom.

Here's a program to illustrate this type of loop:

This program chooses a random number until the user opts to exit, it imports the random module and uses the randint() method.

```
import random


while True:
    input(“Press enter to draw a number “)
    # get a number between 1 to 20
    number = random.randint(1,20)
    print(“You got”,number)
    choice = input(“Do you want to play again?(y/n) “)

    if choice == ‘n’:
        break
```

When you run the program, the loop goes on infinitely until you press ‘n’ which leads to a break statement that ends the loop.

Press enter to draw a number

You got 6

```
Do you want to play again?(y/n) y
Press enter to draw a number
You got 10
Do you want to play again?(y/n) y
Press enter to draw a number
You got 9
Do you want to play again?(y/n) n
>>>
```

Step 17: User-Defined Functions

A function is a block of organized and related statements that is used to perform a specific task. It is a structuring element that allows a code to be used repeatedly in different parts of a program. Using functions enhances program readability and comprehensibility. Functions help make programming more efficient by minimizing repetitions and breaking down long and complex programs into smaller and manageable segments.

Functions are also called methods, procedures, subprograms, routines, or subroutines. There are two types of functions in Python: built-in and user-defined. Built-in functions are those that are provided by Python and are immediately available. User-defined functions are created by users according to Python's syntax.

A function is defined with the **syntax**:

```
def function_name(parameters):
    """docstring"""
    function body
```

Here's an example of a function:

```
>>>def greet(name):
   """Greets the person
   passed as argument"""
print("Hello, " + name + ". Good day!")
>>>
```

It's a good practice to name your function according to the work they perform.

A block of code that defines a function consists of the following parts:

1. def keyword

The keyword ‘def’ defines and names the function and indicates the beginning of the function header.

2. function name

A function is identified by a unique name that is given to it in the function header statement. Function-naming follows the rules set for writing identifiers.

3. parameters

Parameters (argument) are optional and are used to pass values to functions. They are written inside parentheses.

4. colon (:)

A colon marks the end of function headers.

5. docstring

A documentation string or docstring is an optional component that is commonly used to describe what the function does. It is written on the line next to the function header. A docstring can span up to several lines and are enclosed in triple quotes. You can access the docstring with: function_name.__doc__.

For example, this function has a two-line docstring enclosed in triple quotes:

```
>>>def greet(name):
  """"Greets the person
  passed as argument""""
print("Hello, " + name + ". Good day!")
>>>
```

To access the docstring, print the __doc__ attribute of the function ‘greet’.

```
>>> print(greet.__doc__)
Greets the person
  passed as argument
```

```
>>>
```

Likewise, you can access the string through the >>> prompt:

```
>>> greet.__doc__
'Greets the person\n   passed as argument'
>>>
```

6. statement(s)

A function's body consists of one or more valid statements. Multiple statements use the same indentation to form a block.

7. return statement

A return statement is used to return a value from a function. If a return statement is not given inside a function, the function will return the object 'None'.

Here is the **syntax** for the return statement:

```
return [expression_list]
```

This function **lacks a return statement**. Hence, it returns 'None'.

```
>>>def greet(name):
   """Greets the person
   passed as argument"""
print("Hello, " + name + ". Good day!")
>>>print(greet("John"))
Hello, John. Good day!
None
>>>
```

Here's a function **with a return statement**:

```
def absolute_value(number):
    “““Returns the absolute value of
       a number entered by user”””
    if number >= 0:
        return number
    else:
        return -number

print(absolute_value(12))
print(absolute_value(-100))
print(absolute_value(-50))
```

When you run the program, the function will return the absolute value of the given integers:

```
12
100
50
>>>
```

Here is a function with a parameter and return, a function that evaluates if a given number

is an even number. It is a function which returns the number if even, otherwise, function returns a string and None.

```
def even_numbers(number):
   if number % 2 == 0:
       return number
   else:
       print(“Sorry,that’s not an even number.”)

print(even_numbers(22))
22

print(even_numbers(13))
Sorry, that’s not an even number.
None

print(even_numbers(14))
14
```

Here is a **simple function**:

```
def people_id(name):
    “““This function greets the person named as parameter.”””
    print (“Hi, ” + name + “, Welcome and enjoy your stay! Good day!”)

print(people_id(‘Sean’))
print(people_id(‘Kirsten’))
```

```
print(people_id('Dax'))
```

When you run the program, you will get this:

```
Hi, Sean, Welcome and enjoy your stay! Good day!
None
Hi, Kirsten, Welcome and enjoy your stay! Good day!
None
Hi, Dax, Welcome and enjoy your stay! Good day!
None
>>>
```

A function with **an if-elif-else statement**:

```
def member_check(x):
    if x.lower() == "y":
        return("Thanks for your loyalty!")
    elif y.lower() == ("n"):
        return("Please complete membership forms.")
    else:
        return("Please check your response.")
```

```
print(member_check("y"))

print(member_check("n"))

print(member_check("x"))
```

Here's the output:

```
Thanks for your loyalty!
Please complete membership forms.
Please check your response.
>>>
```

Calling a Function

Once it is defined, you can call a function in different ways. You can call it through the >>> prompt, through another function, or through a program.

The simplest way to call a function is through the >>> prompt. You can do this by typing the function name and providing the parameters.

For example, create a simple function that prints a passed string:

```
>>>def stringprinter(str):
print(str)
return;
```

At the prompt, call the stringprinter function and supply the arguments:

```
>>>stringprinter("I'm working but the stringprinter called me!")
I'm working but the stringprinter called me!
>>>
```

Using functions to call another function

In Python, functions can call another function.

For example:

School_sum calls the class_sum function:

```
def class_sum(num):
    return num * 3

def school_sum(m):
    return class_sum(m) + 3

print(school_sum(5))
print(school_sum(8))
print(school_sum(15))
```

Output:

```
17
27
47
>>>
```

Program to Compute for Weighted Average

Students' periodic or final grades are generally computed using the weighted average

method where each criteria for grading is given a certain percentage or weight. In this program, the function get_average calls on another function, ‘average’, to compute for the average grade on each grading criteria. It then applies a given percentage (in decimal format) against the average grade for each criteria and returns the total value. Finally, the program prints the weighted average for each student.

```
Mark = {
    “Name”: “Mark Spark”,
    “Quizzes”: [89.0, 95.0, 78.0, 90.0],
    “Homework”: [89.0, 60.0, 98.0],
    “Recitation”: [89.0,90.0, 88.0],
    “Tests”: [85.0, 92.0]
}
Selen = {
    “Name”: “Selen Jobs”,
    “Quizzes”: [98.0, 100.0, 95.0, 100.0],
    “Homework”: [85.0, 84.0, 90.0],
    “Recitation”: [87.0, 89.0, 90.0],
    “Tests”: [90.0, 97.0]
```

```
}
Shane = {
    “Name”: “Shane Taylor”,
    “Quizzes”: [75.0, 87.0, 95.0, 84.0],
    “Homework”: [92.0, 74.0, 99.0],
    “Recitation”: [80.0, 83.0, 84.0],
    “Tests”: [98.0, 100.0]
}


def average(numbers):
    total=sum(numbers)
    result=total/len(numbers)
    return result

```

```
def get_average(student):

    Quizzes = average(student[“Quizzes”])

    Homework = average(student[“Homework”])

    Recitation = average(student[“Recitation”])

    Tests = average(student[“Tests”])

    print(student[“Name”])

    return .2*Quizzes + .1*Homework + .3*Recitation + .4*Tests



print(get_average(Mark))

print(get_average(Selen))

print(get_average(Shane))
```

The output would be:

```
Mark Spark
87.93333333333334
Selen Jobs
92.28333333333333
Shane Taylor
90.18333333333334
>>>
```

Anonymous Functions

While you would normally define a function with the def keyword and a name, anonymous functions are defined with the lambda keyword and without a name. An anonymous function is variably called a lambda function.

A lambda function has this **syntax**:

```
lambda arguments: expression
```

A lambda function can take as many arguments as you want but it can only have one expression. That single expression is evaluated then returned. You can use a lambda function anywhere a function object is needed.

Here is an example of a program that uses the lambda function that squares the value of input:

```
squared = lambda x: x ** 2

print(squared(10))
```

Output will be:

100

>>>

In the above sample program, lambda x: x ** 2 is the lambda function where x is the argument and x ** 2 is the expression that the interpreter evaluates and returns. The identifier ‘squared’ holds the function object returned by the expression.

The statement which assigns the lambda expression to ‘squared’ is almost like defining a

function named squared with the following statements:

```
def squared(x):
  return x ** 2
```

Lambda functions are commonly used when a nameless function is required on a short-term basis. In Python, lambda functions are generally used as arguments to other functions. It is used with built-in functions such as map() and filter().

Lambda functions with map()

Python’s map() function takes in a list and another function. The function argument is called with all elements in the list and returns a new list containing elements returned from each element in the original list.

Here is an example showing how the map() function is used with the lambda function to get the squared value of every integer in the list.

```
num_list = [1, 3, 5, 7, 9, 11, 8, 6, 4, 2]

squared_list = list(map(lambda x: x ** 2 , num_list))
print(squared_list)
```

The output will be:

```
[1, 9, 25, 49, 81, 121, 64, 36, 16, 4]
>>>
```

Lambda functions with filter()

Python’s filter() function takes in a list and another function as parameters. The function argument is called with all elements in the list and returns a new list containing elements

returned from each element in the original list that evaluates to True.

Here is an example of the usage of the filter() function in a program that filters out odd number from a specified list:

```
num_list = [12, 4, 1, 8, 9, 6, 11, 5, 2, 20]


odd_list = list(filter(lambda x: (x % 2 != 0) , num_list))

print(odd_list)
```

The output will be:

```
[1, 9, 11, 5]
>>>
```

Recursive Functions

Recursion is a programming construct where the function calls itself at least once in its body. The value returned is usually the return value of the function call. A recursive function is a function that calls itself.

While recursion is often associated with infinity, a recursive function has to terminate in order to be used. A recursive function can be brought to an end by downsizing the solution with every recursive call as it moves gradually to a base case. The base case is the condition where a problem can finally be solved without recursion. Recursion can result to an infinite loop if the base case is not reached in the function calls.

The use of recursive function is illustrated in the computation of the factorial of a number. A number's factorial refers to the product of all integers from 1 to the given integer. For example, to find the factorial of 5 (written as 5!): 1*2*3*4*5 = 120. The problem is solved by multiplying the value returned from the previous multiplication operation with the current integer until it reaches 5.

Factorial is implemented in Python using the following code:

```
def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)
```

Run the program and use the print function to find the factorial for the numbers 5 and 7:

```
>>>print(factorial(5))
120
>>>print(factorial(7))
5040
>>>
```

Scope and Lifetime of a Variable

A variable's scope refers to the part of a program where it is recognized. Variables and parameters defined within a function has a scope that is limited within the said function. A variable's lifetime is the period throughout which it exists in memory. The lifetime of a variable within a function coincides with the function's execution. Such variable is destroyed once the return is reached. A function does not have a recollection of a variable's previous values.

Step 18: Python Modules

Any file that contains proper Python code and has the .py extension can be called a Python module. Modules contain statements and definitions. They usually contain arbitrary objects such as classes, functions, files, attributes, and common Python statements like those that initialize the module. These objects can be accessed by importing them. Modules help break down large programs into manageable files. They also promote code reusability. You can, in fact, gather the most frequently used functions, save them in one module, and import them in your other programs.

Modules are known by their main filename, that is, without the .py extension. For example, create a module and save it as multiplier.py.

```
def multiply (a, b):
    product = a * b
  return product
```

The module will then be known as multiplier.

Importing a Module

Importing a module allows us to access the objects, statements, and definitions it contains. There are different ways to import a module.

Through a dot (.) operator

For example, if you want to reuse the multiplier module, you can use the following statements:

```
>>>import multiplier
>>> multiplier.multiply (5, 3)
15
>>>
```

Python's Math Module

Importing the math module allows Python users to access attributes and mathematical functions and constants such as sin() and cosine() functions, pi, and square root. One you import the math module, you can simply place math and a dot before the attribute or function.

To import the math module:

```
>>>import math
>>>
```

The above statement is an example of a generic import where you only import the module without specifying a function. The statement gives you access to math module's entire definition.

To import specific math definitions, attributes, or functions, you will just simply type them after math and a dot. For example:

```
>>>math.pi
3.141592653589793
>>> math.sqrt(100)
10.0
>>> math.gcd(16, 8)
8
>>> math.fabs(-12)
14.0
>>>
```

If you just need a specific function, for instance, square root, it will be tiring to repeatedly type math.sqrt() every time you want to obtain the square root of a number. To avoid this, you can do a function import with this **syntax**:

```
>>> from module import function
```

For example, to import only the square root function:

```
>>> from math import sqrt
```

From here, you only need to type sqrt() and the number:

```
>>> sqrt (25)
5.0
>>> sqrt (100)
10.0
>>> sqrt (81)
9.0
>>>
```

Displaying the Contents of a Module

To list the methods and attributes of a module after importing it, you can use the built-in function dir() and supply the module's name as argument.

For example, to view the contents of the user-defined multiplier module:

```
>>>import multiplier
>>>dir (multiplier)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'multiply']
>>>
```

The math module contains many useful functions. To print the entire content of the math module, you can use the following code:

```
import math
everything = dir(math)
print (everything)
```

This is what you would see on your screen after running the code:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

Getting more information about a module and its function

You can use the help() function to view more information about a module or its specific function.

For instance, if you want to read Python's help documentation for the math function sqrt, you can use the following statements:

```
>>>import math
>>>help(math.sqrt)
```

Once you press enter, Python will show you this:

```
Help on built-in function sqrt in module math:

sqrt(…)
   sqrt(x)

   Return the square root of x.

>>>
```

The Random Module

By importing the random module, you will have access to several functions that are commonly used in games.

Usage of Random Module

The random module is commonly used when you want a program to produce a random number on a specified range. It is used when the program requires the user to pick a random element from a list, roll a dice, pick a card, flip a coin, and similar games.

Random Functions

Random provides the following useful functions:

1. Randint

The randint() function is used to generate a random integer and accepts two parameters. The first one is the lowest number and the second one is the highest number. For example, to generate any integer from 1 to 6:

```
>>>import random
>>>print (random.randint(1, 6))
```

The output will be any one of the integers 1, 2, 3, 4, 5 0r 6.

Here is another example:

```
>>>import random
>>>print(random.randint(0, 100))
96
>>>
```

2. Choice

The choice() function generates a random value from a sequence.

The **syntax** is:

random.choice([‘cat’, ‘dog’, ‘parrot’]).

This function is most commonly used to pick a random item from a list.

```
import random
my_list = [2, 3, 4, 5, 6, 7, 8, 9, 10, “Ace”, “Jack”, “Queen”, “King”]
random.choice(my_list)

print(random.choice(my_list))
print(random.choice(my_list))
print(random.choice(my_list))
```

Run the program and y0u might get the following random output:

```
4
Queen
2
>>>
```

3. Shuffle

The shuffle() function sorts the elements on a list so that they will be arranged in random order.

The **syntax** is:

```
random.shuffle(list)
```

Here’s an example:

```
from random import shuffle
a = [[x] for x in range(15)]
shuffle(a)
```

If you run the program and print a:

```
>>>print(a)
[[14], [5], [10], [7], [11], [13], [0], [2], [9], [1], [6], [12], [3], [4], [8]]
>>>
```

Since numbers are shuffled on a random basis, you will most probably get a different result every time you use the print statement.

Here’s another example of using shuffle():

```
import random
my_list = [‘P’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’, ‘m’, ‘e’, ‘r’]
random.shuffle(my_list)

print(my_list)
```

Here’s one possible output:

```
[‘r’, ‘e’, ‘g’, ‘a’, ‘m’, ‘m’, ‘P’, ‘r’, ‘r’, ‘o’]
>>>
```

4. Randrange

The randrange() function generates a random element from a specified range.

The **syntax** is:

```
random.randrange(start, stop[, step])
```

Here is an example:

```
import random
for x in range(5):
    print(random.randrange(0, 50, 2))
```

Run the program and you'll get 5 random numbers:

```
12
42
48
28
0
>>>
```

Universal Imports

Instead of importing specific object(s), it's possible to import everything from a module by using an asterisk * in the import statement. This is called universal import.

For example:

```
>>> from math import *
>>> sin(4.01) + tan(cos(3.1)) + e
```

```
0.40053169581643777
>>>
```

While it may seem convenient to just import everything instead of explicitly importing a certain function, universal import is not recommended except when using the Python Shell interactively. One reason is the possibility of running several functions with exactly the same name. If you import with an asterisk *, you might risk running functions with obscure origin because you won't know which module they came from.

Importing Several Modules at Once

You can import several modules in one import statement by separating module names with commas. For example:

```
>>>import math, multiplier
>>>
```

When importing a module, you can change the name of the namespace with:

```
import module as module2
```

For example:

```
>>> import math as mathematics
>>>print(mathematics.sqrt(25))
5.0
>>>import multiplier as multi
>>>print("The product of 12 and 3 is ", multi.multiply(12, 3))
The product of 12 and 3 is  36
>>>
```

Step 19: Date and Time

Most programs require date and time data to operate. Python has time and calendar modules that can be used to track times and dates in programs.

To access functions in the time module, you have to import the module with:

```
import time
```

Many time functions return a time value as a tuple of 9 integers. The functions strptime(), and gmtime() also provide attribute names for each field.

For example, if you print **time.localtime()** function with:

```
>>>import time
>>>print(time.localtime())
```

You'll get an output in a tuple format:

```
time.struct_time(tm_year=2016, tm_mon=6, tm_mday=29, tm_hour=23, tm_min=51, tm_sec=29, tm_wday=2, tm_yday=181, tm_isdst=0)
>>>
```

The above time tuple represents the struct_time structure which has the following attributes:

Index	Attributes	Fields	Values
0	tm_year	4-digit year	2016
1	tm_mon	month	1 to 12

2	tm_mday	day	1 to 31
3	tm_hour=23	hour	0 to 23
4	tm_min=51	minute	0 to 59
5	tm_sec	second	0 to 61
6	tm_wday	day of the week	0 to 6 (0 is Monday)
7	tm_yday	day of the year	1 to 366
8	tm_isdst	Daylight savings	0, 1, -1

Formatted Time

You can format time in many different ways with the print function. There are also several ways to get readable time format in Python. A simple way to get time is with the **asctime() function.** Here's an example:

```
>>>import time
>>> time_now = time.asctime( time.localtime())
>>> print (“Current date and time is:”, time_now)
```

The output will be in the following format:

```
Current date and time is: Fri Oct  7 01:35:51 2016
>>>
```

Getting Monthly Calendar

The calendar module provides several methods that allow users to obtain and manipulate monthly and yearly calendars.

To access these methods, you have to import the calendar module:

```
import calendar
```

For example, to get the calendar for the month of July, 2016, you can use the following statements:

```
>>>import calendar
>>>July_cal = calendar.month(2016, 7)
>>>print ("Calendar for the month of:")
>>>print (July_cal)
```

You'll get the following:

```
Calendar for the month of:
   July 2016
Mo Tu We Th Fr  Sa  Su
                 1   2   3
 4   5    6   7   8  9  10
11  12   13  14  15  16 17
18 19   20 21  22 23 24
25 26  27  28  29 30 31
>>>
```

The Time Module

Python's time module provides functions that allow users to work with time and convert between representations. Following is a list of available methods:

time.altzone
time.asctime
time.clock
time.ctime
time.gmtime
time.localtime
time.mktime
time.sleep
time.strftime(fmt[, tuppletime])
time.strptime(string[, format = “%a %b %d %H:%M:%S %Y”])
time.time
time.tzset

The Calendar Module

The calendar module provides functions related to a calendar including print functions that output a calendar for a specified year or month.

Br default, calendar has Monday as the first day of the week. You can change this by calling the **calendar.setfirstweekday() function**.

The calendar module offers the following functions:

calendar.calendar(year, w=2, l=1, c=6)

The calendar.calendar() function returns a multiline string of calendar for a given year.

calendar.calendar(2016, w=2, l=1, c=6)

calendar.firstweekday( )

The function returns the setting for the first weekday of the week. The default starting day is Monday which is equivalent to o.

```
>>>import calendar
>>>calendar.firstweekday()
0
>>>
```

calendar.isleap(year)

This function evaluates if the given year is a leap year. If yes, it returns True. If not, it returns False.

```
>>>import calendar
>>> calendar.isleap(2016)
True
>>> calendar.isleap(2014)
False
>>>
```

calendar.leapdays(y1, y2)

This function returns the sum of leap days within a given range.

```
>>>import calendar
>>> calendar.leapdays(2010, 2016)
1
>>>
```

calendar.month(year, month, w=2, l=1)

This function returns a multiline string of calendar for a given month of the year. The value of w indicates the width (in characters) of the dates while l specifies the allotted lines for each week.

```
>>>import calendar
>>>calendar.month(2016, 7, w=2, l=1)
>>>'     July 2016\nMo Tu We Th Fr Sa Su\n             1  2  3\n 4  5  6  7  8  9 10\n11 12 13 14 15 16 17\n18 19 20 21 22 23 24\n25 26 27 28 29 30 31\n'
>>>
```

Use the print function to get a calendar for the month in a more user-friendly format.

```
>>>print(calendar.month(2016, 7, w=2, l=1))
   July 2016
Mo Tu We Th Fr Sa Su
             1  2  3
4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
>>>
```

calendar.monthcalendar(year, month)

This function returns a list with sublists of integer elements. Each sublist represents a week. Days outside of the given month are represented as zero. Days of the month are set as numbers corresponding to their day-of-month.

```
>>>import calendar
>>>calendar.monthcalendar(2016, 7)
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10], [11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24], [25, 26, 27, 28, 29, 30, 31]]
>>>
```

calendar.monthrange(year, month)

This function returns two integers. The first integer denotes the weekday code (o to 6) for the first day of the month in a given year. The second indicates the number of days in the

specified month (1 to 31).

```
>>>import calendar
>>>calendar.monthrange(2016, 7)
(4, 31)
>>>
```

calendar.prmonth(year, month, w=2, l=1)

It returns a calendar for the month like what you'll see if you use print (calendar.month(year,month,w,l)).

calendar.setfirstweekday(weekday)

Sets the first day of the week using a weekday code that represents the days from Monday (0) to Sunday (6).

calendar.weekday(year, month, day)

This function provides the weekday code for a specified date. The parameters are year, month (1 to 12 for the 12 months of the year), and day (o to 6 from Monday onwards).

```
>>>import calendar
>>> calendar.weekday(2016,6,30)
3
>>>
```

Datetime

Python has another method, datetime(), that you can use to retrieve current system date and time. Before you can use it, you'll have to import the module **datetime.now().**

For example, enter the following on the >>>prompt:

```
>>>from datetime import datetime
>>>datetime.now()
datetime.datetime(2016, 7, 8, 2, 35, 38, 337769)
>>>
```

In order to convert the date and time information into a more readable format, you can

import ‘strftime’ from Python’s standard library with these statements:

```
>>>from time import strftime
>>>strftime(“%Y-%m-%d %H:%M:%S”)
```

Here’s the result:

```
‘2016-07-08 02:35:02’
```

Step 20: Namespaces

A name or identifier is a way of identifying and accessing objects. For instance, when you enter an assignment statement like x = 5, 5 is the object stored in memory while x is the name that we use to refer to the object. You can get the memory address of some objects by using **id()**, a built-in function.

For example:

```
>>> x = 5
>>>id(x)
1518352208
>>> id(5)
1518352208
>>>
```

The above results with the id() functions tells you that x and 5 refer to the same object.

Here are other expressions that can tell how Python responds to reassignments:

The use of id() function provides the RAM address of x.

```
>>> x =5
>>>id(x)
1518352208
>>>
```

Here, x gets reassigned and holds the value of 5 + 7. It is now in another memory location:

```
>>> x = 5 + 7
>>> id(x)
1518352320
>>>
```

Finally, 5 is assigned to ‘y’. Take note that using id() on both the name and object returns the same memory location of the object 5, that is, at 1518352208. This makes Python programming efficient since you won’t have to create duplicate copies of the object.

```
>>>y = 5
>>> id(5)
1518352208
>>> id(y)
1518352208
>>>
```

Since a function is also an object, you can use a name to refer to it. You will then use the function’s name to call it.

For example:

```
def greetings():
    print("Good Morning!")

x = greetings
```

Run the program and use x() to call the function x.

```
>>> x()
Good Morning!
>>>
```

Now that you have a better idea about names and what you can use them for, you’re ready to learn the concept of namespaces. A namespace is basically a collection of names. It is a

mapping of every name that has been defined to corresponding objects. For instance, Python aut0matically creates a namespace that has all the built-in names when you start the interpreter. Hence, built-in functions are always available. Each Python module creates a global namespace for itself. A module can have several classes and functions. A local namespace is created every time a function is called.

Scope

Although namespaces have distinct names, it is not always possible to access all of them from just about anywhere in the program. There is such a thing as scope, a program component that allows users to access a namespace directly without a prefix. There are at least 3 nested scopes:

- the current function's scope with local names
- the module's scope with global names
- the outermost scope with built-in names

Every time a reference is made inside a function, the interpreter searches for the name in the following hierarchy: 1. local namespace 2. global namespace and 3. built-in namespace.

Here is an example showing how scope and namespace works in Python:

```
def main_function():
    x = 50
    def sub_func():
        y = 15

z = 5
```

In the above code, the variable x is within the local namespace of main_function(). The

variable y is in the local namespace of sub_function, a nested function. The variable z, on the other hand, is in the global namespace.

Inside the main_function, x is a local variable, y is nonlocal, while z is global. When in the main_function, you can read and assign new values to x, a local variable but you can only read the variable y from sub_function. If you try to reassign y to a new value, that will be stored in the local namespace you're in, the main_function, and will be an entirely different variable from the variable y in the sub_function.

The same thing happens if you try to reassign z, the global variable, inside the main_function. If you want all references and assignments to go to the global variable, you have to declare it as global with the expression 'global z'.

To illustrate, here is a program that uses and accesses the variable z in three different namespaces:

```
def main_function():
    z = 10
    def sub_function():
        z = 20
        print('z =',z)

    sub_function()
    print('z =',z)

```

```
z = 33
main_function()
print('z =',z)
```

If you run this program, the output will be:

```
z = 20
z = 10
z = 33
>>>
```

Here's a program where all references and assignments go to the global variable z:

```
def main_function():
    global z
    z = 10
    def sub_function():
        global z
        z = 20
        print('z =',z)
```

```
    sub_function()
    print('z =',z)

z = 33
main_function()
print('z =',z)
```

The output will show the value assigned to z as a global variable in the nested function:

```
z = 20
z = 20
z = 20
>>>
```

Step 21: Classes and Object-Oriented Programming

Python is an object-oriented programming language. This means that it emphasizes working with data structures called objects. This is in contrast with procedure-oriented languages which are focused on functions.

An object can refer to anything that could be named such as functions, integers, strings, floats, classes, methods, and files. It is a collection of data and the methods that utilize data. Objects are flexible structures that can be used in different ways. They can be assigned to variables, dictionaries, lists, tuples, or sets. They can be passed as arguments.

A class is a data type just like a list, string, dictionary, float, or an integer. When you create an object out of the class data type, the object is called an instance of a class.

In Python, everything is an object — and that includes classes and types. Both classes and types belong to the data type ‘type’. The data value that you store inside an object is called an attribute while the functions associated with it are called methods. A class is a way of creating, organizing, and managing objects with like attributes and methods. Designing an object requires planning and decision-making on what the objects will represent and how you can group things together.

Defining a Class

To define a class, you will use the keyword class followed by a class identifier and a colon. If the new class is a child class, you’ll have to enclose a parent class inside the parentheses. A class name starts in uppercase by convention. The class definition is usually followed by a docstring which provides a short description of the class.

This is an example of a simple class definition:

```
class Members:
    I created a new class.
    pass
```

This is an example of a definition of a class that takes an object:

class Employees (object)

I have just defined a class with an object.

    pass

The docstring of a class can be accessed with:

ClassName.__doc__.

When you use the keyword ‘class’ to define a class, this tells Python to create a new class object with the same name. Defining a class creates a namespace that contains the definition of all the attributes of the class, including special attributes that start with double underscores. This object can then be used to access the attributes of the class and to create or instantiate new objects of the class.

For example, create a new class named MyClass:

```
class MyClass:
    “This is a new class.”
    b = 12
    def greet (self):
        print (‘Good morning!’)
```

To access the attributes of MyClass:

```
>>> MyClass.b
```

```
12
>>> MyClass.greet
<function MyClass.greet at 0x02CA5D20>
>>> MyClass.__doc__
'This is a new class.'
>>>
```

Creating an Object

The class object can be used to create instances of the new class. Creating an object is no different from making a function call at the >>> prompt. For example, the following statement creates an instance object called obj:

```
>>>obj = MyClass()
```

To access the attributes of an object, you can use the object name as prefix before the dot. An object's attribute is either method or data. An object's method refers to the corresponding functions of the class. A function object defines a method for objects created from a class. For example, create a class named NewClass:

```
class NewClass:
    "This is a new class."
    x = 5
    def func (self):
        print("I'm a function object.")
```

To access the function attribute of NewClass:

```
>>> NewClass.func
```

```
<function NewClass.func at 0x02BB5D20>
>>>
```

Create a new object from NewClass:

```
>>>obj = NewClass()
```

NewClass.func, a class attribute, is a function object because it defines a method for all object created from NewClass. Hence, obj.func is a method object.

```
>>> obj.func
<bound method NewClass.func of <__main__.NewClass object at 0x02BB1C10>>
>>>
```

You probably notice the parameter ‘self’ in function definition inside classes. Yet, when the method obj.func was called without an argument, Python went on with the process and did not raise an error. That is because the object itself is passed as the first argument. In Python, when an object calls its method, the object automatically becomes the first argument. By convention, it is called ‘self’. You can use any name but for uniformity, it is best to stick with the convention. If you need to place more arguments, you have to place ‘self’ as the first argument.

The __init__() method

The __init__() method is a class constructor which is used to initialize the object it creates. Whenever a new instance of the class is created, Python calls on this initialization method, a special method with double underscores prefix. The method takes at least one argument, the ‘self’, to identify each object being created.

Examples:

```
class Achievers:
   def __init__(self) :
```

```
class Achievers (object):
   def __init__(self, name, salary) :
```

A method is a function used in a class. Hence, the __init__() function is called a method when it is used to initialize objects of a class.

Instance Variables

Instance variables are used to link all instantiated objects within the class. They are required if you have to use multiple arguments other than ‘self’ in the initialization method.

For example:

```
class Employees:
   “Common base for all employees.”
   counter = 0
   def __init__(self, name, position, salary) :
      self.name = name
      self.position = position
      self.salary = salary
```

The class definition code block states that whenever an instance of the Employees class is created, each employee will have a copy of the variables initialized using the __init__ method.

You can instantiate members of the class Employees with these statements:

```
>>>emp_1 = Employees (“Chuck”, “supervisor”, “5000.00”)
>>>emp_2 = Employees (“Kurt”, “encoder”, “3500.00”)
>>>emp_3 = Employees (“Charlie”, “proofreader”, “4000.00”)
```

Now, use the print function to see the connection between the initialized variables and members' variables:

```
>>>print(emp_1.name, emp_1.position, emp_1.salary)
>>>print(emp_2.name, emp_2.position, emp_2.salary)
>>>print(emp_3.name, emp_3.position, emp_3.salary)
```

You should see the following output:

```
Chuck supervisor 5000.00
Kurt encoder 3500.00
Charlie proofreader 4000.00
```

Following is a longer code that illustrates class definition and the use of the datetime module:

```
import datetime


class Member:


    def __init__(self, firstname, surname, birthdate, country, email, telephone):
        self.firstname = firstname
        self.surname = surname
        self.birthdate = birthdate
```

```
        self.country = country
        self.email = email
        self.telephone = telephone

    def age(self):
        now = datetime.date.today()
        age = now.year - self.birthdate.year
        if now < datetime.date(now.year, self.birthdate.month, self.birthdate.day):
            age -= 1
        return age

member1 = Member(
    “Jessica”,
    “Law”,
    datetime.date(1995, 6, 1), # year, month, day
```

```
    “United States”,
    “jane.law@yow.com”,
    “811 432 0867”
)

member2 = Member(
    “Martin”,
    “Donz”,
    datetime.date(1991, 8, 16), # year, month, day
    “Canada”,
    “martindonz@jobs.com”,
    “580 324 7111”
)
```

```
print(member1.firstname)
print(member1.surname)
print(member1.birthdate)
print(member1.country)
print(member1.email)
print(member1.telephone)
print(member1.age())

print(member2.firstname)
print(member2.surname)
print(member2.birthdate)
print(member2.country)
print(member2.email)
print(member2.telephone)
print(member2.age())
```

Run the code and you'll see the following output onscreen:

Jessica

Law

1995-06-01

United States

jane.law@yow.com

811 432 0867

21

Martin

Donz

1991-08-16

Canada

martindonz@jobs.com

580 324 7111

24

You will notice that two functions were defined inside the Member class body. These functions are the object's methods.

The first is a special method, __init__(). Whenever you call the class object, it creates a new instance of the class and Python immediately executes the __init__() method on the new object. The __init__ method is used to pass on to the new object and to set it up with all the data and parameters that you have specified. The second method, the 'age' method, calculates the member's age with the use of current date and the birthdate.

Take note that the __init__ function creates attributes and sets them according to the values passed in as parameters. Although it's not compulsory, the attributes and parameters are usually given the same name.

Adding an attribute

It is possible to add an attribute outside of the __init__ function.

To illustrate, here is a program that has two numbers as class attributes:

```
class NumberPairs:
    def __init__(self, first = 0, second = 0):
        self.first = first
        self.second = second

    def display_data(self):
        print("The first number is {0} and the second number is {1}.".format(self.first,
self.second))
```

The following program runs will show some class behaviors:

To create object ‘a’ with 2 arguments:

```
>>> a = NumberPairs (3, 10)
>>> a.display_data()
The first number is 3 and the second number is 10
>>>
```

This one will create object ‘b’ with only one number, 6, as argument:

```
>>> b = NumberPairs(6)
>>> b.display_data()
```

The first number is 6 and the second number is 0.

>>>

Notice that it took the supplied argument as the first number and zero as the default argument for the second number.

If you want to add a new attribute to the object ‘b’, you can easily do so with the syntax ‘obj.attr = value’.

>>>b.attr = 16

To display the values of the number pair and the new attribute:

>>>b.first, b.second, b.attr

(6, 0, 16)

>>>

The change only affected the object b and not a. Hence:

>>> a.attr

Attempting to access the non-existent object a.attr will raise the AttributeError:

Traceback (most recent call last):

  File “<pyshell#7>”, line 1, in <module>

    a.attr

AttributeError: ‘NumberPairs’ object has no attribute ‘attr’

Deleting Objects and Attributes

An object’s attribute can be deleted with the use of the keyword del. The **syntax** is:

del obj.attribute

To illustrate, create a new object ‘pairs2’ under the class ‘NumberPairs’:

```
>>> pairs2 =NumberPairs(5, 10)
```

To delete pairs2:

```
>>>del pairs2.second
```

If you try to call the function display_data on pairs2, Python will throw an AttributeError:

```
>>> pairs2.display_data()
Traceback (most recent call last):
…
AttributeError: ‘NumberPairs’ object has no attribute ‘second’
```

The same thing happens when you delete the function display_data:

```
>>> del NumberPairs.display_data
>>> a.display_data()
Traceback (most recent call last):
  File “<pyshell#4>”, line 1, in <module>
    a.display_data()
NameError: name ‘a’ is not defined
>>>
```

To delete the object itself, you will this **syntax**:

```
del object
```

Using the above example, you can delete object ‘a’ with:

```
>>>del a
>>>
```

While the above statements may sound so simple, what happens internally is more complicated. When you create an instance object with c = NumberPairs(16, 7), the name c binds with the object in memory. When you issue the command del c, the relationship is broken and the name c is removed from its corresponding namespace. The object, however, remains in memory. It is automatically destroyed later if it not referenced to by another name. This process is commonly known as ‘garbage collection’.

Modifying Variables within the Class

It is quite possible to modify variables belonging to the same class. This feature becomes handy when you have to change the value that a variable stores based on what occurs inside a class method.

To illustrate, this program shows how the value stored in the variable ‘condition’ changed from ‘new’ to ‘used’ as the program is executed:

```
class Dryer(object):
    condition = "new"
    def __init__(self, brand, model, color):
        self.brand = brand
        self.model  = model
        self.color = color
    def show_dryer(self):
        return "This is a " + self.color + " " + self.brand + " " + self.model + " dryer."
    def use_dryer(self):
        self.condition = "used"
my_dryer = Dryer("Ecolux", "Natural", "silver")
print(my_dryer.show_dryer())
print(my_dryer.condition)
my_dryer.use_dryer()
print(my_dryer.condition)
```

When you run the program, you will have the following output:

```
This is a silver Ecolux Natural dryer.
new
used
>>>
```

Inheritance

Inheritance is a programming structure that allows a new class to inherit the attributes of

an existing class. The new class is called the child class, subclass, or the derived class while the class it inherits from is the parent class, superclass, or base class. Inheritance is an important feature because it promotes code reusability and efficiency. If you have, for example, an existing class that delivers what your program requires, you can create a subclass that will take most of what the existing class does, enhance its functionality, or perhaps partially override some class behavior to make it a perfect fit for your purpose.

To define a class that will inherit all functions and variables from a parent class, you will use the **syntax**:

```
class ChildClass(ParentClass):
```

To illustrate, here's a program that creates a new class Students:

```
class Students(object):
    “common base for all members”
    def __init__(self, name, average, level):
        self.name = name
        self.average = average
        self.level = level

stud1 = Students(“Michelle Kern”, 88.3, “Senior”)
 stud2 = Students(“Jennifer Pauper”, 93.5,  “Junior”)
```

```
stud3 = Students(“Dwight Jack”, 90.0,  “Junior”)


print(stud1.name, stud1.average, stud1.level)
print(stud2.name, stud2.average, stud2.level)
print(stud3.name, stud3.average, stud3.level)
```

When you run the program, you’ll get the following output:

```
Michelle Kern 88.3 Senior
Jennifer Pauper 93.5 Junior
Dwight Jack 90.0 Junior
>>>
```

Assuming you want to create a new class, Graduates, which will inherit from the Students class and use a new attribute, grad_year, here’s how your program might look:

```
class Graduates(Students):
    “common base for Graduates members”
    def __init__ (self, name, average, level, grad_year):
        Students.__init__(self, name, average, level)
        self.grad_year = grad_year

```

```
mem1 = Graduates(“Randy Doves”, 88.2, “graduate”, “2015”)

mem2 = Graduates(“Dani Konz”, 85.0, “graduate”, “2014”)

mem3 = Graduates(“Ashley Jones”, 89.5, “graduate”, “2016”)

print(mem1.name, mem1.average, mem1.level, mem1.grad_year)

print(mem2.name, mem2.average, mem2.level, mem2.grad_year)

 print(mem3.name, mem3.average, mem3.level, mem3.grad_year)
```

When you run the program, you would have the output:

```
Randy Doves 88.2 graduate 2015
Dani Konz 85.0 graduate 2014
Ashley Jones 89.5 graduate 2016
>>>
```

Multiple Inheritance

The above section discussed the concept of single inheritance where a child class inherits from a single parent class. Python also supports multiple inheritance. Multiple inheritance is a programming feature which allows one class to inherit methods and attributes from two or more parent class. Python offers a well-structured and sophisticated approach to multiple inheritance.

The **syntax** for a class definition where there are more than one parent class is:

```
class MultiDerivedClassName (ParentClass1, ParentClass2, ParentClass3, …):
pass
```

Multilevel Inheritance

A multi-level inheritance allows a new class to inherit from a derived class. In Python, multilevel inheritance can go as deep as you want it to go. Following is the syntax for multi-level inheritance:

```
class Base:
    pass

class Derived_1(Base):
    pass

class Derived_2(Derived_1):
    pass
```

class Derived_3(Derived_2)

Step 22: Python Iterators

An iterator is any Python objectwhichcan be iterated onto return data on a per itembasis. Python iterators can be found in many places. They are implemented in comprehensions, ‘for’ loops, and generators but are not readily seen in plain sight. Iterator objects must performthemethods __iter__() and __next__().

In general, built-in Python containers like string, list, and tupleare iterable because they can be used to produce an iterator. The function iter() is used to return an iterator by calling the __iter_() method. The function next() is then used to iterate through each element. The StopIteration exception is raised whenever the end is reached.

To illustrate:

First define num_list:

>>>num_list = [5, 8, 2, 4]

Then use iter() to generate an iterator:

>>>num_iter = iter(num_list)

Then, view the date type of num_iter:

>>>num_iter

<list_iterator object at 0x03300F30>

Now, iterate through num_list:

>>>next(num_iter)

5

>>>next(num_iter)

8

>>>next(num_iter)

2

```
>>>next(num_iter)
4
>>>
```

You can get similar results by using theexpression ‘obj.__next__()’. To illustrate, create a new iterator, new_iter to iterate through the num_list:

```
>>>num_list = [5, 8, 2, 4]
>>> new_iter = iter(num_list)
```

Then, view the data type of new_iter:

```
>>>new_iter
<list_iterator object at 0x03A50FD0>
>>>
```

Finally, iterate through each item in the list:

```
>>> new_iter.__next__()
5
>>> new_iter.__next__()
8
>>> new_iter.__next__()
2
>>>new_iter.__next__()
4
>>>
```

Iterating through containers need not always be as tedious. The use of the ‘for loop’, offers a more sophisticated way to iterate automatically through an object.

For example:

```
>>>for item in num_list:
```

```
    print (item)
```

5

8

2

4

The use of the ‘for loop’ in the above example makes iteration a simple task. Internally, however, a ‘for loop’ is executed as an infinite loop with try and except statements. The following example will illustrate how Python executes a ‘for loop’.

This is the statement you would normally see:

```
for item in x_iterable:
```

Internally, Python implements the above statement block as an infinite while loop:

```
iterator_object = iter (x_iterable)

while True:
  try:
item = next(iterator_object)
  except StopIteration:
    break
```

Behind the scenes, the ‘for loop’ calls iter() on x_iterable to create an iteratorobject.In addition, the next() function is used within the ‘while’ structure to return the succeeding item. After exhausting all items, Python will raise a StopIteration exception which indicates the end of the loop. The StopIteration is the only exception that can terminate the loop. All other exceptions are ignored.

Creating a Python Iterator

To build an iterator, you need to execute the iterator protocol, namely, the__iter__() and __next__() methods.

The method __iter__() is used to return an iterator object. The __next__() method is then

called to return the next element in the series. Once the end is reached and in succeeding calls for the next item, the method shouldprompt a StopIteration message.

To illustrate, here is a class that will implement an iterator of multiples of 3:

```
File  Edit  Format  Run  Options  Window  Help
class MultThree:
    """A Class that implements an iterator
    of multiples of 3"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.m = 0
        return self

    def __next__(self):
        if self.m <= self.max:
            result = 3 ** self.m
            self.m += 1
            return result
        else:
            raise StopIteration
```

Here is an iterator that will iterate through the multiples of three:

```
>>>x = MultThree(6)
>>> n = iter(x)
>>>next(n)
1
>>>next(n)
3
>>>next(n)
9
>>>next(n)
27
>>>next(n)
81
>>>next(n)
```

```
243
>>>next(n)
729
>>>next(n)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in<module>
    next(i)
    File "C:/AppData/Local/Programs/Python/Python35-32/Mult_Three.py", line 18, in
__next__
raise StopIteration
StopIteration
```

Step 23: Python Generators

Building your own iterator can be a tedious and lengthy process. You will have to implement a class with the iterator protocol and raise StopIteration when you've come to the end of the file. This counter intuitive steps can be avoided when you use generators.

A Python generator is a much simpler way of buildingan iterator. By returning an object that you can iterate over, a generator automatically manages the processes required to create an iterator.

Python generatorsare functionswith a yield statement. To create a generator function, you have to define it in the same way that you would define a regular function. The inclusion of at least one yield statement makes a function a generator function. A yield expression temporarily halts a function as it retains local variables and goes on to implement successive calls.

To illustrate, here is a generator function that contains multiple yield statements:

```
def gen_function():
    """"define generator function""""
    x = 5
    print("This is the initial value.")
    # The gen_function has multiple yield expressions
    yield x

    x += 5
```

```
print("This is the second value.")

yield x



x += 5

print("This is the final value.")

yield x
```

This is how the code will go when you run it interactively:

First, the function does not execute instantl:

```
>>>g = gen_function()
```

You can use next() to iterate through the elements:

```
>>>next(g)
This is the initial value.
5
```

Local variables are retained between consecutive calls:

```
>>> next(g)
This is the second value.
10
>>>next(g)
This is the last value.
```

```
15
>>>
```

Once the function ends, it raises StopIteration automatically on subsequent calls.

You will notice that unlike normal functions where local variables are destroyed as the function returns, the local variables in a generator function are retained between calls. In addition, you can only iterate over a generator object once. You will have to define a new generator object if you want to restart the iteration process. That is, you’ll make another assignment such as x = gen_function.

Step 24: Files

File is a named disk location which is used to store related data. You will usually want to reuse your data in the future and to do so, you have to store them in memory through a file. In Python, file management is done using a file object.

The File Object Attributes

A file object from an opened file can provide a number of important information about that file.

Here are the attributes associated to the file object:

Attribute	**Returns:**
file.name	the name of the file
file.mode	the access mode used to open the file
file.closed	True if close, False if open

For example:

label = open(“names.txt”, “w”)
print (“Filename: “, label.name)
print (“Closed or Not : “, label.closed)
print (“Opening Mode : “, label.mode)

```
label.close()
```

You will get the output:

```
Filename:  names.txt
Closed or Not :  False
Opening Mode :  w
>>>
```

File Operations

There are 4 basic file-related operations in Python:

- opening a file
- reading from a file
- writing to a file
- closing a file

The Open() function

The open() function facilitates the opening of a file for reading, writing or both. It creates a file object, also called a handle, that you can use to call other methods that can be used with the function. Its **syntax** is:

```
file object=open(filename [, access_mode][, buffering])
```

The filename is the string that represents the name of the file that you want to access.

The access mode is an optional parameter that allows you to indicate the mode in which you want to access the file. Options include read, write, or append. You can also specify whether you want the file to be opened in either binary or text mode. The text mode supports working with strings while the binary mode returns bytes and allows access to non-text files such as images or exe files. The default access modes are read mode and text mode:

Buffering allows users to specify their preferred buffering. No buffering takes place when the value is at zero. When the buffering value is one, line buffering is done as you access files. When the value is greater than one, the buffering process is implemented according to the given buffer size. A negative value tells Python to implement the system default.

The following modes are available when accessing files in Python:

Modes for Accessing Files in Text Format

r	Read mode (default): opens a file for reading.
w	Write museode: overwrites an existing file or creates a new file if the file does not exist.
r+	Read and write mode.
w+	Read and write mode: overwrites an existing file or creates a new one if the file does not exist.
a	Append mode: adds data at the end of the file, creates a new file if the file does not exist.
a+	Read and append mode: adds data at the end of the file, creates a new file if the file does not exist.
x	Opens file for exclusive creation, fails if the file already exists.

Modes for Accessing Files in Binary Format:

rb+	Read and write mode.
wb+	Read and write mode: overwrites an existing file or creates a new file if the file does not exist.
ab+	Read and append mode: adds data at the end of the file or create a new file if the file does not exist.

Examples:

Open file in read mode, the default mode:

```
>>>f = open("access.txt")
```

Open file in write mode:

```
>>>f = open("access.txt",'w')
```

Writing to a File

If you want to modify or write something into a file, you have to open the file in append mode, write mode, or in exclusive creation mode. You should exercise caution when choosing the mode as specifying a wrong mode can easily cause your file and the data stored to be erased or overwritten.

Closing a File

When you're done with your work, it's important to close your files properly to free up the resources used and to avoid accidental deletion or modification. Closing your file properly is a way of telling Python that you're done with your editing or writing work and that it is time to write data to your file.

Here is the **syntax** for opening and closing a file:

```
fileobject = open("trial.txt",encoding = 'utf-8') # open trial.txt file

fileobject.close()
```

Opening, Writing to, and Closing a Text File

The following activity will illustrate Python's file management system. To start, create a new file with the statement:

```
>>>f = open("newfile.txt", "w")
```

```
>>>f.close()
>>>
```

By opening the file "newfile.txt" in "w" (write) mode, you are telling Python to overwrite any data stored in the file, if any. This also means that you'll be starting over with an empty file. The file was closed properly using the close() method.

To build your file, open the newfile.txt again using "w" mode:

```
>>>f = open("newfile.txt","w")
```

Use the write() method to write the following strings on the file:

```
>>> f.write("A file can hold important information.")
38
>>> f.write("You can use it to tell people about your program.")
49
>>> f.write("Files are objects, too, and you can use them in your programs.")
62
>>> f.write("It can hold different types of information  - abc, 123, 1.2, 4.5, &*$#")
70
>>>
```

Take note that the Python Shell returns a number every time you enter a statement with the write() method. The number refers to the number of characters written to a file.

When you're done writing to the file, don't forget to close the file with the close() method:

f.close()

Reading a Python File

There are several ways to read a text file in Python. Here are the most common ones:

- with the readlines() method
- with 'while' statement
- with an iterator

- with the ‘with statement’

The readlines() method

The readlines() method is an easy way to read and parse each line in a text file. You have to use the readlines() method on the file object to tell Python to read the entire text file. After that, you’ll create a variable to store all lines from the text file. To view the file, you’ll either use the print() function on the variable or simply type the variable name on the >>> prompt to view the lines. To illustrate, you can open the newfile.txt you have written previously:

Open the ‘newfile.txt’ on a read only mode:

```
>>>f = open(‘newfile.txt’, “r”)
```

Create a variable, ‘lines’ that will store text lines from the readlines() method:

```
>>>lines = f.readlines()
```

Type lines to view the contents of the text file:

```
>>> lines
[‘A file can hold important information.’]
>>>
```

Don’t forget to close the file:

```
>>>f.close()
>>>
```

You probably noticed that the write() method does not use line breaks and simply writes whatever you type and enter. Hence, what you got in the above example is a paragraph. If you want to write your strings on a line by line basis, you can try encoding a new line character, “\n” at the end of each string. For example:

```
>>> f.write(“I want this string to be an independent statement.\n”)
>>>
```

The readlines() method is a simple an great way to go over the lines of a small text file. When it comes, however, to reading large files, you're bound to experience issues on memory efficiency. Large text files are better handled by methods that facilitate line by line reading. There are two ways to do this: by using the while loop and the for loop.

To illustrate both methods, you need to create a text file that will store data on a per line basis using a new line character at the end of a string.

Create a new file 'linefile' on write mode:

```
>>> f = open("linefile", "w")
```

Write line by line strings on the file:

```
>>> f.write("This is a new text file.\n")
25
>>> f.write("It holds several lines of strings.\n")
35
>>> f.write("Writing them line by line makes file operations more efficient.\n")
64
>>> f.write("Did you notice the integer returned as you enter a line?\n")
57
>>>
```

Close the file:

```
>>> f.close()
```

Line by Line Reading of Text Files with the 'while' loop

Here is a simple loop that you can use to read a Python file on a per line basis:

Open the linefile.txt on read only mode:

```
f = open('linefile.txt', ')
line = f.readline()
```

Keep reading line one at a time until file is empty:

```
while line:
    print line
    line = f.readline()
f.close()
```

When you run the program, here’s what you should see on the Python Shell:

This is a new text file.

It holds several lines of strings.

Writing them line by line makes file operations more efficient.

Did you notice the integer returned as you enter a line?

Line by Line Reading of Text Files using an Iterator

Another way to read Python text files one line at a time is with the use of an iterator. Here is a ‘for’ loop to iterate through the linefile.text you have used earlier:

```
f = open(‘linefile.txt’)
for line in iter(f):
    print line
f.close()
```

The Python Shell should display the lines one by one:

This is a new text file.

It holds several lines of strings.

Writing them line by line makes file operations more efficient.

Did you notice the integer returned as you enter a line?

The 'with statement'

The 'with statement' can be used to read through the lines of a file. The 'with structure' allows you to open a file safely and let Python close it automatically without using the file.close() method.

Example:

The following code creates newfile.txt in write mode, writes 4 lines of text, and closes the file:

```
fileobject = open("newfile.txt", 'w')
fileobject.write("A file can store text and images.")
fileobject.write("You can open an image file with the binary mode.")
fileobject.write("Binary stream objects lack encoding attribute.")
fileobject.write("Files contain bytes.")
fileobject.close()
```

To read each line in the file using the 'with pattern', you can use the following statements:

```
line_num = 0
with open('newfile.txt', 'r') as f_file:
    for line in f_file:
        line_num += 1
        print('{:>3} {}'.format(line_num, line.rstrip()))
```

The 'with pattern' makes use of the 'for loop' to read the files line by line. This shows that the file object is an iterator that can produce a line when you require a value. The 'line' variable holds the entire line content including carriage returns.

You can use the format() string method with the print() function if you want your program to print out each line and the corresponding line number. You can even specify how the line number should appear by using format specifiers. In the above code, the specifier {:>3} tells Python to print the argument (line_num for line number) right-justified inside 3

spaces. The rstrip() method is used to remove trailing whitespaces as well as carriage return characters.

If you run the program, here's what the output would be:

1 A file can store text and images.

2 You can open an image file with the binary mode.

3 Binary stream objects lack encoding attribute.

4 Files contain bytes.

Appending Data to a File

To append data to the file "linefile.txt", you have to reopen the file on append mode with 'a':

```
f = open("linefile.txt","a") #opens the "linefile.txt" file
f.write("Did you guess how easy it is to append data to a saved text file?\n")
f.write("Appending data to a file is as easy as replacing "w" with an "a"\n.")
f.close()
```

Apply the iterator above to see what's on linefile.txt:

```
f = open('linefile.txt')
for x in iter(f):
    print(x)
f.close()
```

You should see the updated linefile.txt:

This is a new text file.

It holds several lines of strings.

Writing them line by line makes file operations more efficient.

Did you notice the integer returned as you enter a line?

Did you guess how easy it is to append data to a saved text file?

Appending data to a file is as easy as replacing “w” with an “a”.

Renaming a File

Python’s os module offers some methods that can be used to perform various file operations such as deleting and renaming files.

Before you can use the os module, you’ll need to import it with the syntax:

import os

Importing the os module gives you access to a range of useful methods.

The rename() method

The rename() method is used to rename an existing file. It takes two arguments and uses the following syntax:

```
os.rename(current_filename, new_filename)
```

Example:

import os

#Rename a file from file1.txt to file2.txt

os.rename(“file1.txt”, “file.txt”)

Deleting a File

To delete an existing file, you can use the remove() method which, like rename(), can be called from the os module. The remove() method takes only one argument and has the following **syntax**:

```
os.remove(file_name)
```

Example:

```
import os

# Delete file2.txt
os.remove("file2.txt")
```

Binary Files

Files are not limited to text. They may also contain images or binary files.

To open binary files, you will use the same syntax you use for opening text files except that this time, you have to use the binary mode which is denoted by the letter 'b'.

For example, to open an image stored in drive c:

```
>>>mypict = open('c:/doggie.jpg', 'rb')
```

The file object of a file opened in binary mode possess most of the attributes of a file object created from opening a text file. It has the mode and name attributes. Unlike a text file, however, it has no encoding attribute. That's because binary files, unlike text files, require no conversion. What you save into it is exactly what you will get.With a binary file, you will be reading bytes, in contrast to reading strings in a text file. Python will raise an AttributeError if you try to use a non-existent encoding attribute.

The following examples will illustrate the attributes of a binary file:

```
>>> mypict.name
c:/doggie.jpg'

>>> mypict.mode
'rb'
```

When you use the read() method in binary mode, any number you put in as a parameter

will refer to the amount of bytes that would be read.

To illustrate, this statement opens an image file beach.jpg in binary read mode and creates a file object mypict:

```
>>> mypict = open(‘c:/beach.jpg’, ‘rb’)
```

The next statement uses the read() method on mypict and supplies 5 as argument.

```
>>> info = mypict.read(5)
```

To access the data stored in info:

```
>>> info
b’\xff\xd8\xff\xe0\x00’
```

Use type to see the data type stored in info:

```
>>> type(info)
<class ‘bytes’>
```

The tell() method returns the number of bytes that has been read. Since you have specified 5 bytes to read, Python returns the same number of bytes.

```
>>> mypict.tell()
5
```

```
>>> mypict.seek(0)
0
```

To read the entire image file, use the read() method on mypict with no arguments:

```
>>> info = mypict.read()
```

You can enter info on the >>>prompt to view the contents of the entire file but that will probably be a pageful of strange letter combinations. Instead, you can use len() to see the total number of bytes read and stored in the variable info:

```
>>> len(info)
3893
```

File Methods

Python has several file methods that can be used to handle files. The most commonly used methods are summarized below:

File.writer(str)

Writes stringo to a file

The **syntax** is:

```
fileobject.write(str)
```

Example:

```
>>> fileobj = open("my_file.txt", "w")
>>> fileobj.write("I am a new member.\n")
19
>>> fileobj.close()
```

File.writelines(sequence)

Writes a sequence of strings

The syntax is:

```
fileobject.writelines(sequence)
```

Example:

```
#This program opens the my_file.txt and appends a sequence of strings.

fileobj = open("my_file.txt", "r+")
print ("Filename: ", fileobj.name)

sequence = ["The club now has 100 members.\n", "I am the 100th member."]

#Write sequence at the end of my_file.txt
fileobj.seek(0, 2)
lines = fileobj.writelines(sequence)

# Read the updated file from the start.
fileobj.seek(0,0)
for index in range(3):
   lines = next(fileobj)
   print("Line No %d - %s" % (index, lines))

# Close file
fileobj.close()
```

This will be the output:

```
Filename:  my_file.txt
Line No 0 - I am a new member.

Line No 1 - The club now has 100 members.

Line No 2 - I am the 100th member.
```

File.readline(size)

Reads an entire line from file

The**syntax** is:

fileobject.readline(size)

Size is an optional parameter that you can use if you want to specify the number of bytes that should be read.

Example:

Assuming that you have created a text file named alphabet.txt and that you have performed a write operation on the file to write several lines:

```
>>> fileobj = open("alphabet.txt", "w")
>>> fileobj.write("There are 26 letters in the alphabet.\n")
38
>>> fileobj.write("A is the first letter of the alphabet.\n")
39
>>> fileobj.write("There are 5 vowels and 21 consonants.\n")
38
>>> fileobj.write("Z is the last letter.\n")
22
>>> fileobj.close()
```

The following program will read lines from the file:

```
# Open a file.
fileobj = open("alphabet.txt", "r+")
print ("Filename: ", fileobj.name)

lines = fileobj.readline()
print ("Read Line: %s" % (lines))

lines = fileobj.readline(14)
```

```
print ("Read Line: %s" % (lines))

lines = fileobj.readline()
print ("Read Line: %s" % (lines))

lines = fileobj.readline(18)
print ("Read Line: %s" % (lines))

# Close file
fileobj.close()
```

The output will be:

```
Filename:  alphabet.txt
Read Line: There are 26 letters in the alphabet.

Read Line: A is the first
Read Line:  letter of the alphabet.

Read Line: There are 5 vowels
```

File.readlines()

Reads until end-of-file using readline()

The **syntax** is:

```
fileobject.readlines(sizehint)
```

The readlines() method returns a list that contains the lines. By default, it reads until EOF but it has an optional argument, sizehint, that can be used to instruct Python to read whole lines approximating the given sizehint bytes. It returns an empty string when the EOF is reached immediately.

Example:

The file myfile.txt contains the following lines:

A dictionary is mutable.
A tuple is immutable.
A string is immutable.
A list is mutable.
Strings, lists, and tuples are sequence types.

This program will show how the method readlines() is used:

```
# Open myfile
fileobject = open(“myfile.txt”, “r+”)
print (“Filename: “, fileobject.name)

lines = fileobject.readlines(2)
print (“Read Line: %s” % (lines))

lines = fileobject.readlines()
print (“Read Line: %s” % (lines))

# Close file
fileobject.close()
```

Here’s the result when you run the code:

```
Filename:  myfile.txt
Read Line: [‘A dictionary is mutable.\n’]
Read Line: [‘A tuple is immutable.\n’, ‘A string is immutable.\n’, ‘A list is mutable\n’, ‘Strings, lists, and tuples are sequence types.\n’]
```

Here's what happens when you tweak the code and use an argument on the first readlines():

```
# Open myfile
fileobject = open("myfile.txt", "r+")
print ("Filename: ", fileobject.name)

lines = fileobject.readlines()
print ("Read Line: %s" % (lines))

lines = fileobject.readlines(3)
print ("Read Line: %s" % (lines))
```

The output would be:

```
Filename:  myfile.txt
Read Line: ['A dictionary is mutable.\n', 'A tuple is immutable.\n', 'A string is immutable.\n', 'A list is mutable\n', 'Strings, lists, and tuples are sequence types.\n']
Read Line: []
```

File Positions: file.tell() and file.seek

File.tell()

Returns present position of file pointer

The **syntax**is:

```
fileobject.tell()
```

Example:

The alphabet.txt file described in the preceding example contains the following lines:

There are 26 letters in the alphabet.
A is the first letter of the alphabet.
There are 5 vowels and 21 consonants.
Z is the last letter.

The following program illustrates the usage of the tell() method:

```
fileobj = open("alphabet.txt", "r+")
print("Filename: ", fileobj.name)

lines = fileobj.readline()
print("Read Line: %s" % (lines))

pos = fileobj.tell()
print("Current Position: " ,pos)

# Close file
fileobj.close
```

Here's the output:

```
Filename:  alphabet.txt
Read Line: There are 26 letters in the alphabet.

Current Position:  39
```

File.seek()

Sets file's current locatopm at offset

The **syntax**is:

fileobject.seek(offset[, from])

offset

position of file read/write pointer

from

From is an optional parameter which has zero(0) as default. There are three possible values for whence:

0 – absolute file positioning

1 – seek relative to current position

2 – seek relative to the end of the file

Take note that if you open a file for appending with either ‘a’ or ‘a+’, any seek() operation is nullified on the next write operation. If you open a file for writing in append mode with ‘a’, the method is not operable. On the other hand, if you open a file in text mode with ‘t’, the only valid offsets are those returned by tell().

The seek()method returns none.

Example:

The following lines are stored in newfile.txt:

This is line 1 of 5.

This is line 2 of 5.

This is line 3 of 5.

This is line 4 of 5.

This is line 5 of 5.

This program illustrates the usage of seek():

Open a file

```
fileobj = open("newfile.txt", "r+")
print ("Filename: ", fileobj.name)

lines = fileobj.readlines()
print ("Read Line: %s" % (lines))

# Set the pointer to the start of the file
fileobj.seek(0, 0)
lines = fileobj.readline()
print ("Read Line: %s" % (lines))

# Close file
fileobj.close()
```

Run the program and you will get these results:

```
Filename:  newfile.txt
Read Line: ['This is line 1 of 5.\n', 'This is line 2 of 5.\n', 'This is line 3 of 5.\n', 'This is line 4 of 5.\n', 'This is line 5 of 5.\n']
Read Line: This is line 1 of 5.
```

File.read(n)

Reads size bytes from a file

The syntax is:

fileobject.read(size)

Size is the number of bytes that will be read.

Example:

The file myfile.txt contains the following lines:

A string is a sequence.
Strings and tuples are immutable.
A list is a mutable sequence.
Python offers many file methods.
The os module has useful methods for directories.

This program shows how the read() method is used to display piecemeal string:

```
# Open a file
fileobj = open("myfile.txt", "r+")
print ("Filename: ", fileobj.name)


lines = fileobj.read(35)
print ("Read Line: %s" % (lines))

# Close file
fileobj.close()
```

Run the program and you'll get the following results:

```
Filename:  myfile.txt
Read Line: A string is a sequence.
Strings and
```

File.truncate([size])

Resizes the file

The **syntax**is:

fileObject.truncate([size])

The truncate() method truncates the size of the file. It has an optional argument for size which, if given, will be used as the basis for truncating the file. If size is not provided, the default is the current position. The truncate() method will not work for files which are opened in ‘read only’ mode.

Example:

This example makes use of the file linefile.txt which contains the following lines:

This is the first line.

The second line is longer than the first line.

This is the third line.

The fourth line is not important.

This is the fifth line and it is the longest line of all.

This code will show how the truncate() method is used:

```
fileobject = open(“linefile.txt”, “r+”)
print (“Filename: “, fileobject.name)

lines = fileobject.readline()
print (“Read Line: %s” % (lines))

fileobject.truncate()
lines = fileobject.readlines()
print (“Read Line: %s” % (lines))

# Close file
fileobject.close()
```

This would be the output:

Filename: linefile.txt
Read Line: This is the first
Read Line: []

File.flush()

Flushed the internal buffer

The **syntax** is:

```
fileobject.flush()
```

This program illustrates the use of the flush() method:

```
# Open a file
fileobject = open("onefile.txt", "wb")
print ("Filename: ", fileobject.name)

# The method() returns none but can be called with the read operations
fileobject.flush()

# Close file
fileobject.close()
```

This is what the output will be:

Filename: onefile.txt

File.close()

Closes an open file

The syntax is:

fileobject.close()

The following illustrates the use of the close() method:

```
# Open a file
fileobject = open(“myfile.txt”, “wb”)
print (“Filename: “, fileobject.name)

# Close file
fileobject.close()
```

File.isatty()

Returns True it file if attached to a try to a tty or similar device and False if otherwise.

The syntax is:

fileobject.isatty()

Example:

This code illustrates the usage of the file.isatty() method:

```
# Open a file
fileobject = open(“myfile.txt”, “wb”)
print (“Filename: “, fileobject.name)

val = fileobject.isatty()
print (“Return value : “, val)
```

Close file
fileobject.close()

Since the file is not attached to any tty or tty-like device, the output will be:

Filename: myfile.txt
Return value : False

File.fileno()

Returns integer file descriptor

The operating system keeps tab of all opened files by creating a unique entry to represent each file. These entries are called file descriptors and are represented by integers.

The syntax is:

```
fileobject.fileno()
```

Example:

```
# Open a file
fileobject = open("myfile.txt", "wb")
print ("Filename: ", fileobject.name)

filenum = fileobject.fileno()
print ("File Descriptor: ", filenum)

# Close file
fileobject.close()
```

This results to the following:

Filename: myfile.txt

File Descriptor: 3

file.readable()	returns True if file is readable
file.writable()	returns True if file is writable
file.detach()	separates and returns binary buffer from TextIOBase

Step 25: Handling Errors or Exceptions

Errors are quite common in programming and they fall into two major types: syntax errors and runtime errors.

Syntax Errors

Errors caused by non-compliance with the language structure or syntax are called syntax or parsing errors.

These are the common syntax errors:

- misspelled keywords
- incorrect indentation
- omission of a required keyword or symbol
- placement of keywords in wrong places
- an empty block

Examples:

```
if num >= 100      #a colon is required in if statements
    print (“You passed the exam!”)

for n in num[4, 19, 25, 50, 26, 75]      #print statement should have been indented
if x == 50:                              #because it is inside the if block
print(“Enter a password!”)
```

Runtime Errors

Errors that occur during runtime are called exceptions. If unsolved, a runtime error may cause a program to crash or terminate unexpectedly. Common runtime errors include IndexError (for index which is out of range), KeyError (when key is not found) or ImportError (for missing import module). Python creates an exception object every time it encounters an exception and displays a traceback to the error if it’s not managed.

Here are examples of exceptions:

- accessing a non-existent file
- using an identifier that has not been defined
- performing operations on incompatible data types
- division by zero
- accessing a non-existent element on a dictionary, list, or tuple

Built-in Exceptions

Python has several built-in exceptions and they are raised whenever a runtime error occurs:

Built-in Exception	Raised when:
IndexError	a specified index is out of range
KeyError	a dictionary key is not found
ZeroDivisionError	the divisor is zero
IndentationError	indentation is not correct
EOFerror	the input() function reaches end-of-file condition
ImportError	imported module is not available
AssertionError	assert statement fails
NameError	a variable is not found in local/global scope
FloatingPointError	floating point operator fails
UnicodeError	a Unicode-related encoding/decoding error occurs
AttributeError	an attribute or a reference assignment fails

Catching Exceptions

There are different ways to catch and manage an exception in Python:

try and except

One way of managing a Python exception is by using “try and except statements”. Critical parts of the program that are likely to cause exceptions are placed within the ‘try’ clause while the exception code is given inside the ‘except’ clause.

For example:

```
try:
    num = int(input(“Please enter a number between 0 and 15: “))
    print(“You entered %d.” % num)
except ValueError:
    print(“That’s not a number! Please try again.”)
```

Python will try to execute all statements in the ‘try’ block. If a ValueError is encountered, control immediately passes to the ‘except’ block. Any unprocessed statement in the ‘try’ block will be ignored. In this case, a ValueError exception was given to handle non-numeric responses.

try…finally

A try statement can also include a “finally” clause. The “finally” clause is always executed regardless of the actions done about the program error. It is also used to clean and release external resources.

```
try:
    num = int(input(“Please enter a number between 0 and 15: “))
    print(“You entered  %d.” % num)
except ValueError:
    print(“That’s not a number! Please try again.”)
else:
    print((“That will be your meal stub number.”)
finally:
    print((“Great! Don’t forget to claim your meals.”)
```

Python Cheat Sheets

To help with your learning you can download and print this Cheat Sheet from our website. We offer two different ways to download it;

- You can open it through Microsoft Word, edit it as you wish and print it.
- Or you can also download it as it is and print it.

To do so, simply go to our website at www.jthompsonbooks.com -> Bonus -> Python’s Companion. You can also <click here> to be automatically directed to it. Our website will ask you for a password.

The password is:

#ilovepython

Please note that the password is in 1 word in lowercase.

Variable Assignment

string = “string”
integer = 1
list = [item1, item2, item3, …]
tuple = (item1, item2, item3, …)
dictionary = { key1 : value1, key2 : value2, key3 : value3 … }
mutli_line_string = “““ multi-line string”””

unicode_string = “unicode string”
class_instance = ClassName (init_args)

Accessing Variable Values

value = string[start:end]	example: “string”[1:4] → “tri”
value = dictionary[key]	example: my_dict[key1]
value = dictionary.get(key, default_value)	
value = list[start:end]	example: [2, 7, 5, 8][1:2] → [7]
value = list[index]	example: [4, 9, 6, 8][1] → 9
value = ClassName.class_variable	example: MyClass.age
value = class_instance.function(args)	
value = class_instance.instance_variable	

Python Operators

Arithmetic Operators

Addition	x = a + b	5 + 10 → 15
Subtraction	x = a – b	15 – 10 → 5
Multiplication	x = a * b	5 * 10 → 50

Division	x = a / b	15 / 3 → 5
Modulos	x = a % b	11 % 3 → 2
Floor Divison	x = a // b	11 // 3 → 3

Assignment Operators

=	assignment	Assigns the value of the right operand to the left operand.	c = a + b assigns value of a + b into c
+=	add and	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	subtract and	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	multiply and	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	divide and	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
**=	exponent and	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	floor division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

%=	Modulus and	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

Relational Operators

2 < 10	is less than
5 > 3	is greater than
4 <= 2+2	is less than or equal to
2 * 5 >= 2 * 3	is greater than or equal to
10 ==2 * 5	is equal to
18 != 2 * 6	is not equal to

Logical Operators

or	(12 > 9) or (15 < 10) → True	(4 > 14) or (8 < 5) → False
and	(10 > 2) and (15 > 5) → True	(5 > 2) and (12 > 20) → False
not	not (6 > 3 ** 5) → True	not (6 < 4 ** 4) → False

Identity Operators

Verifies if the left and right operands refer to the same memory location:

is	a = 8, b = 8	a is b → True
is not	x = 5, y = 10	a is not b → True

Membership Operators

Checks if a given value occurs in a sequence:

in	x = [“a”, “b”, “c”, “d”, “e”]	“b” in x → True
not in	x = “immutable”	‘u’ not in x → True

Bitwise Operators

Assuming x = 15 and y = 10				
&	Bitwise AND	x & y → 10	bin(10) → ‘0b1010’	
|	Bitwise OR	x | y → 15	bin(15) → ‘0b1111’	
^	Bitwise XOR	x ^ y → 5	bin(5)	→ ‘0b101’
~	Bitwise NOT	~ x → -16	bin(-16) → ‘-0b10000’	
<<	Bitwise left shift	x << 2 → 60	bin(60) → ‘0b111100’	
>>	Bitwise right shift	y >> 2 → 2	bin(2) → ‘0b10’	

Strings

s = ‘new string’	Creates ‘new string’ in single quotes.
s = “new string”	Creates “new string” in double quotes.
s = “““Hello, world”””	Creates a string “““Hello,world””” in triple quotes.
y=x.replace(“old”,“new”,1)	Replaces 1 st occurrence of “old” string with “new string”.

n = len(str)	Stores the length (no. Of characters) of a string to n.
“abc” in str	Checks if a given substring exists in a string .
“%s%s” % (“prog”, “ram”)	→ ‘program’
print(“string”)	→ string
str.split(“delim”, limit)	“a/t”.split(“/”) → [‘a’, ‘t’]
str.upper()	Returns a view of a string in uppercase.
str.lower()	Returns a view of a string in lower case.
str.title()	Returns a view of a string where the first letter of every word is capitalized, the rest are in lowercase.
str.count(“a”)	Sum the occurrence of a character or series of character in a string.
str.find(“abc”)	Searches for a character or series of characters, returns the index number.
str.isalpha()	Checks if a string is alphabetic.
str.isalnum()	Checks if a string is alphanumeric.
str.isidentifier()	Checks if a string is a valid identifier.
str1.join(seq)	Concatenates a string with a sequence of string.

str.lstrip()	Returns the string without leading characters or whitespaces.
str.rstrip()	Returns the string without trailing characters or whitespaces.
str.strip()	Returns the string without leading and trailing characters or whitespaces.
str1.index(str2, beg, end)	Checks the occurrence of a given substring and returns its index.
str1.rindex(str2, beg, end)	Searches for the last occurrence of a given substring and returns its index.
str.zfill(width)	Returns a string with leading zeros within the given width.
str.rjust(width[, fillchar])	Returns right-justified string within a given width, has optional fill character.
str.ljust(width[, fillchar])	Returns left-justified string within a given width, takes optional fill character.
str.center(width[, fillchar])	Returns center-justified string within a given width, takes optional fill character.
str.endswith(“suffix”)	Checks whether a string ends in a given suffix.
str.startswith(“prefix”)	Checks whether a string ends in a given prefix.
string1 + string2	“prog” + “ram” → ‘program’

Lists

L = [“a”, 1, “X”, 4.5,]	list creation
L[0]	access the first element
L[0:2]	access the first two elements
L[-4:]	access the last four elements
L[1:4] = [“e”, 10, “o”]	replace elements on index 1 to 3 with given values
del L[3]	remove list element on given index
len(L)	returns the number of items on a list
max(L)	returns the largest item on a list
min(L)	returns the smallest item on a list
sum(L)	returns the total value on list
list()	converts an iterable to a list
L.append(a)	adds a given value to list L
L.extend(L2)	adds elements in L2 to L, same as L3 = L + L2
L.remove(a)	a is a value you want to remove from list L
L.sort()	sort a list containing elements of the same type
L.count(“e”)	searches for a value and returns number of occurrence
L.reverse()	sorts a list in the reverse (descending) order
L.pop()	simple stack, removes the last element
L.index(a)	index of first occurrence of a given value

a in L	membership test: does L contain a? True or False
[x*3 for x in L if x>4]	list comprehension

Tuple

x = 1, 2, 3, 4, 5	
x = (1, 2, 3, 4, 5)	
x = (1, 2, 4, “a”, “b”, “c”)	
x = (“book”, (1, 3, 40), [8, 4, 7])	
x = (“student”,)	
x = ()	
1, 2, 3, 4, 5 = x	
x[3]	# access element on index 3
x[4:7]	# access element on indices 4 to 6
del x	# delete tuple x
‘g’ in my_tuple	# membership test, returns True or False
mytuple.count(a)	returns the number of elements equal to given value
mytuple.index(a)	returns index of the first element equal to given value

len(mytuple)	returns the number of elements in a tuple
max(mytuple)	returns the largest element in a tuple
min(mytuple)	returns the smallest element in a tuple
sorted(mytuple)	returns a view of a sorted tuple
sum(mytuple)	returns the total of all items in a tuple
tuple()	converts an iterable to a tuple

Dictionary

D = {‘key1’: 15, ‘key2’: ‘xyz’, ‘key3’: 78.50 }	creates a dictionary
D = dict(‘key1’=15, ‘key2’=‘xyz’, ‘key3’=78.50)	creates a dictionary
D = {}	creates an empty dictionary
keys = (‘x’, ‘y’, ‘z’)	
dict2 = dict(pairs)	creates dict2 from a list of tuple key-value pairs
D = dict.fromkeys(keys)	create new dictionary with no values
for k in D: print(k)	iterates over dictionary keys

my_dict[‘Age’]	accesses the value of the given key
my_dict.get(‘Name’)	uses the get() method to access a value
my_dict[‘Age’] = 24	adds a new key and value to a dictionary
my_dict[‘Age’] = 20	modifies the value of a given key
my_dict.pop(‘Name’)	removes the given key and its value
my_dict.popitem()	method that removes a random key-value pair
my_dict.clear()	removes all key-value pairs from dictionary
del my_dict	deletes a dictionary
dict1.update(dict2)	updates dict1 with key-value pairs from dict2
dict.items()	returns a list of a dictionary’s key-value pairs
dict.values()	returns a list of all dictionary values
dict.keys()	returns a list of dictionary keys
dict.setdefault(‘a’, None)	searches for a given value in dictionary
dict_2 = dict_1.copy()	creates a copy of dict_1 dictionary

squares={x:x*3 for x in range(5)}	dictionary comprehension
newdict = dict.fromkeys(L, 3)	takes items in sequence and uses them as keys
3 in a_dict	dictionary membership test
len(a_dict)	returns the number of items in a dictionary
for x in dict.values(): print(x)	prints values
for x, y in dict.items():	key-value tuples
list(dict.keys())	dictionary keys in list form
sorted(dict.keys())	sorted dictionary keys in list form

Sets

my_set = {2, 4, 6, 8, 10}	
my_set = {‘x’, ‘y’, ‘z’, a’, ‘e’}	
my_set = set()	
my_set = {10.0, “Python”, (9, 7, 5), 5}	
Set_a = {2, 4, 6} List_1 = [2, 5, 4, 1, 6] S = set(List_1) # set ([1,2,4,5,6])	

set([5,4,3, 1])	creates set from a list
my_set.add(12)	adds an element to a set
my_set.update([1, 3, 5])	
my_set.discard (‘x’)	removes an element from a set
my_set.remove (‘f’)	removes an element from a set
my_set.pop()	removes a random item and returns it
my_set.clear()	removes all elements of a set
X | Y	union: combines the elements of X and Y
X.union(Y)	combines the elements of X and (
x & y	intersection: combines common elements of X and Y
x.intersection(y)	method that combines common elements of X and Y
x – y	difference: set of elements found in x but not in y
x.difference(y)	returns a set of elements found in x but not in y
x ^ y	set of elements that are not common in x and y
b.symmetric_difference(a)	

‘s’ in my_set	set membership test
len(my_set)	returns the number of elements on a set
max(my_set)	returns the largest element on a set
min(my_set)	returns the largest element on a set
sorted(my_set)	returns a sorted view of a set
sum()	returns the total of all items in a set

Loops

for num in range(4):	# 4 numbers starting from zero: 0,1,2,3
for num in range(0, 10, 3):	# start, end, progression: 0,3,6,9
for num in range(0, 9):	# start, end: 0,1,2,3,4,5,6,7,8
for a, z in dict.letters():	# dict[key1] = 50 dict[key2] = 100
print(“dict[{}]={}”.format(a,z))	
num_list = [2, 4, 6, 8]	
for a, z in zip(List_1, List_2):	# return tuple pairs
for index, value in enumerate(num_list):	# index, value

for x in sorted(set(num_list)):	# set sorted from num_list
for val in reversed(num_list):	# reverse sort of num_list

Conditional Statements

if…else

```
if a > 90:
    print("Good job!")
else:
    print("Please try again!")
```

if…elif…else

```
if x > 20:
    print("Sorry, you have exceeded the limit.")
elif x > 12:
    print("You have a chance to join the jackpot round.")
else:
    print("You may roll the dice again.")
```

Built-in Functions

range(n1, n2, step)	creates a list with arithmetic progression
input()	x = input(“Enter a number: “) takes user’s input
print(x)	prints the given value
abs(n)	returns absolute value of integers or floats
max(x)	returns the largest value among two or more numeric data
min(x)	returns the smallest value among two or more numeric data
type()	returns the data type of given argument

User-Defined Functions

To define a function

def function(parameters):
    “““docstring”””
    function body

def adder(num):
    ”““Function that adds numbers”””
    function body

Classes

To define a class:

class ClassName:
    “““This is a new class.”””
    b = “apple”
    def greet(self):
    print(‘Hello, World!’)

To access ClassName’s attributes:

ClassName.b → “apple”

ClassName.greet → <function ClassName.greet at 0x02AD5D20>

ClassName.__doc__ → ‘This is a new class.’

To create an object from ClassName:

obj = ClassName()

Files

f = open(“trial.txt”, “w”)   # open options: r , r+, rb, rb+, w, wb

f.write(“I’m a string.\n”)

f.close()

L = open(“trial.txt”).readlines()   # returns list of lines

for line in open(“trial.txt”): print(line, end=””)

Text File Opening Modes

r	Default mode, read mode.
w	Write mode, creates new file or overwrites existing file.
r+	Read and write mode.
w+	Read and write mode, creates new file or overwrites existing file.
a	Append mode: if the file exists, it adds at the end of the file; if file does not exist, it creates a new file.
a+	Read and append mode: if the file exists, it reads and appends data at the end of the file; if file does not exist, it creates a new file.
x	Opens file for exclusive creation, fails if the file already exists.

Binary File Opening Mode

rb+	Binary format read and write mode: file pointer is located at the start of the file.

wb+	Binary format write mode: overwrites an existing file, creates a new file if non-existent.
ab+	Binary format read and append mode: if the file exists, it adds data at the end of the file; if the file does not exist, it creates a new file.

File Operations

fileobj = open(“myfile”, “w”)	creates fileobj, opens myfile on write mode
fileobj.write(“string”)	writes ‘string’ to file
fileobj.writelines(“I am me./n”)	writes a sequence of strings
fileobj.readline()	reads an entire line, takes an optional argument for byte size
fileobj.readlines()	reads all lines until EOF, takes an optional sizehint argument
fileobj.tell()	returns the position of the file pointer
fileobj.seek(offset[from])	sets file pointer’s current position at offset
fileobj.read(n)	reads size bytes from file
fileobj.truncate(size)	truncates the file, takes an optional agreement for size
fileobj.flush()	flushes the internal buffer
fileobj.close()	closes an open file
fileobj.isatty()	checks if file is attached to a tty device
fileobj.fileno()	return an integer file descriptor

Date and Time

import time

formatted time:

time_now=time.asctime(time.localtime(time.time()))

→ Thu Jun 30 01:22:15 2016

from datetime import datetime

datetime.now()

from time import strftime

strftime(“%Y-%m-%d %H:%M:%S”) → ‘2016-07-08 02:35:02’

Python Modules

import module	Imports a Python module.
import math	Imports the math module.
math.sqrt(n)	Math.sqrt(100) → 25
math.gcd(n1, n2)	Math.gcd(12, 14) → 2
math.fabs(n)	Math.fabs(-15) → 15.0
import random	Imports Python’s random module.
random.randint(1, 12)	returns a random integer from 1 to 12.
random.choice([‘a’, 1, 2])	Returns a random value from a given list.
random.shuffle(([1, 2, 3])	Rort and arranges items in a list in a random order

import time	Imports Python’s time module.
time.localtime()	Returns time in tuple format.
import calendar	Imports Python’s calendar module.
calendar.month(2016, 7)	Imports calendar for July 2016.
calendar.calendar(year, w=2, l=1, c=6)	Imports calendar for the year.
calendar.firstweekday( )	Sets first day of the week, default is Monday.

Help!

Oh no, you’re stuck! Chances are that someone else had the same problem you’re having and has written about it online. Here are some great ressources you can use to find answers to your questions.

Google

You can simply google the exact error message you are getting. For example:

```
>>> a.attr
```

Attempting to access the non-existent object a.attr will raise the AttributeError:

```
Traceback (most recent call last):
  File “<pyshell#7>”, line 1, in <module>
    a.attr
AttributeError: ‘NumberPairs’ object has no attribute ‘attr’
```

Simply google “Attribute Error” to find ways to resolve the issue.

FAQ

- Python.org offers many answers to many common general questions. To read about Python Frequently Asked Questions <click here>.

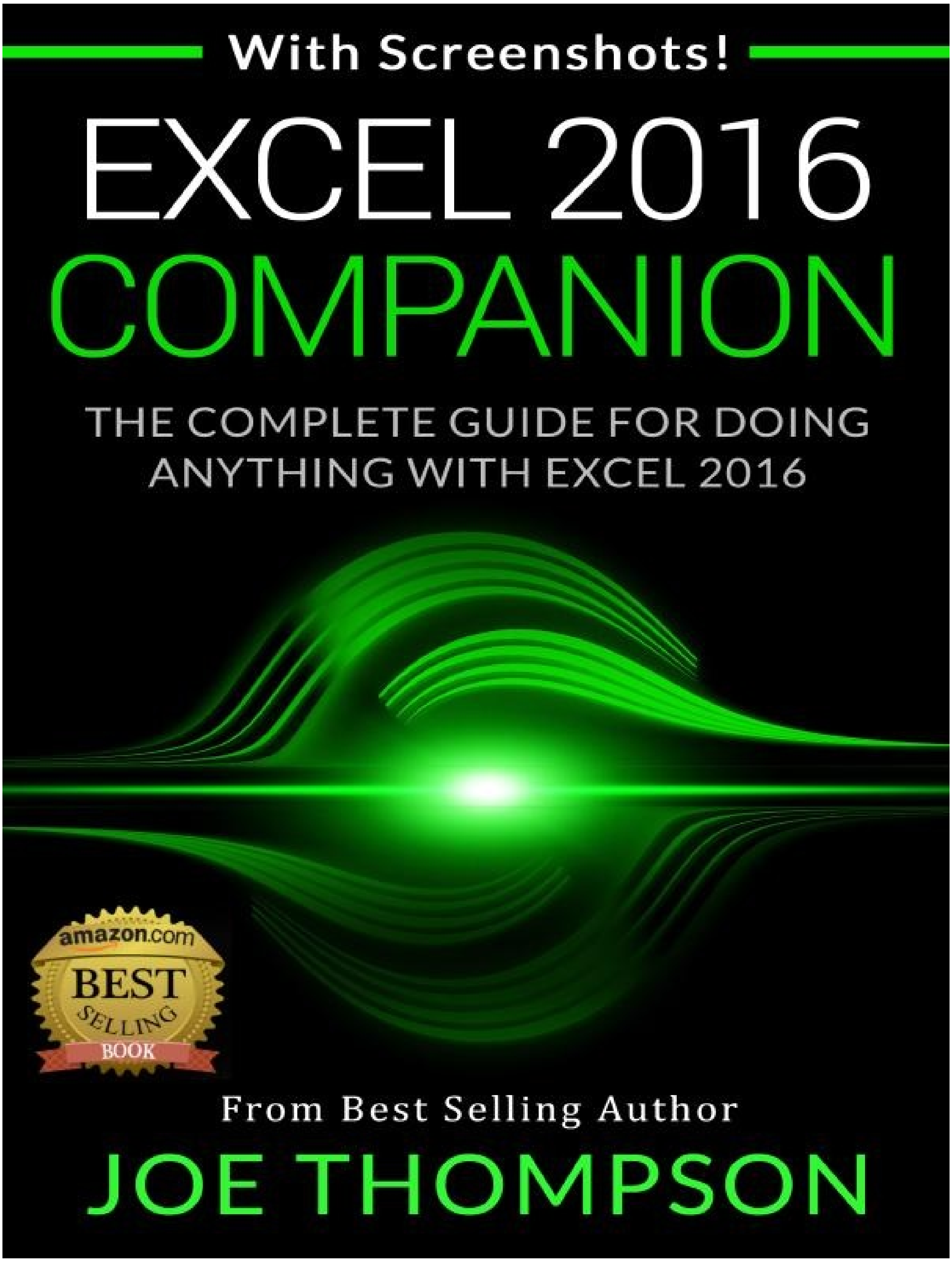
- Stack Overflow (www.stackoverflow.com) is one of the most popular question-and-answers sites for programmers. Some members post their questions when they are stuck and others try to help them with answers.

IRC (Internet Relay Chat)

If you are stuck and searching online isn’t proving helpful, then you might try asking someone in an IRC channel. IRC is a chat system where people around the world communicate live. You can create your own account by going to

www.webchat.freenode.net. To join the main Python channel, you must enter /join #python in the input box.

Other Best Selling Books You Might Like!



Read this Step by Step book to easily understand Excel 2016 in no time! Click <here> to see my book and click on it's cover to see the first pages!

Conclusion

Thank you again for downloading this book!

I hope that this book was able to help you to learn the fundamentals of Python programming quickly and easily. By now, I know that you can make useful, practical programs that will automate many of your daily tasks at work and at home.

The next step is to take up advanced Python programming courses that can help you create more powerful programs.

If you happened to like my book please leave me a quick review on Amazon. I personally read every single comment on the site. Your feedback is very important to me as it will help improve the book and ensure your satisfaction.

Thank you!

Joe Thompson

Table of Contents

Introduction

An Overview of Python

Step 1: Installing Python

Installing Python in Windows

Which version should I use?

Installing Python in Mac

Running the Installation file

Starting Python

IDLE versus the command line interface (CLI)

IDLE

The Command Line Interface (CLI)

Different ways to access Python’s command line

If you’re using Windows

If you’re using GNU/Linux, UNIX, and Mac OS systems

Step 2: Working with IDLE

The Python Shell

The File Menu

The Edit menu

The Shell Menu

The Debug Menu

The Options Menu

The Window Menu

The Help Menu

Writing your First Python Program

Accessing Python’s File Editor

Typing your code

Saving the File

Running the Application

Exiting Python

Step 3: Python Files and Directories

The mkdir() Method

The chdir() Method
The getcwd() Method
The rmdir() Method
Step 4: Python Basic Syntax
Python Keywords (Python Reserve words)
Python’s Identifiers
Five rules for writing identifiers
A Class Identifier
Naming Global Variables
Naming Classes
Naming Instance Variables
Naming Modules and Packages
Naming Functions
Naming Arguments
Naming Constants
Using Quotation Marks
Statements
Multi-line statements
Indentation
Comments
Docstring
Step 5: Variables and Python Data Types
Variables
Memory Location
Multiple assignments in one statement
Assignment of a common value to several variables in a single statement
Data Types
Boolean Data Type
Step 6: Number Data Types
Integers (int)
Normal integers
Octal literal (base 8)

Hexadecimal literal (base 16)
Binary literal (base 2)
Converting Integers to their String Equivalent
integer to octal literal
integer to hexadecimal literal
integer to binary literal
Floating-Point Numbers (Floats)
Complex Numbers
Converting From One Numeric Type to Another
To convert a float to a plain integer
To convert an integer to a floating-point number
To convert an integer to a complex number
To convert a float to a complex number
To convert a numeric expression (x, y) to a complex number with a real number and imaginary number
Numbers and Arithmetic Operators
Addition (+)
Subtraction (-)
Multiplication (*)
Division (/)
Exponent (**)
Modulos (%)
Relational or Comparison Operators
Assignment Operators
= Operator
add and +=
subtract and -=
multiply and *=
divide and /=
modulos and %=
floor division and //=
Bill Calculator
Built-in Functions Commonly Used with Numbers

abs(x)
max()
min()
round()
Math Methods
Math.ceil(x)
Math.floor(x)
Math.fabs()
Math.pow()
Math.sqrt()
Math.log()
Step 7: Strings
Accessing Characters in a String
String Indexing
The Len() Function
Slicing Strings
Concatenating Strings
Repeating a String
Using the upper() and lower() functions on a string
Using the str() function
Python String Methods
The replace() method
Case Methods with String
Upper()
Lower()
Swapcase()
Title()
Count() method
The find() method
Isalpha()
Isalnum()
Isidentifier()

The join() method

Lstrip() method

Rstrip() method

Strip([chars])

Rfind() method

Index() method

Rindex() method

Zfill() method

Rjust() method

Ljust() method

Center() method

Endswith() method

Startswith() method

Iterating Through a String

Step 8: Output Formatting

The print() function

Using the str.format() method to format string objects

Other Formatting Options

‘<’

‘>’

‘^’

‘0’

‘=’

Step 9: Lists

Accessing Elements on a List

Indexing

Negative Indexing

Slcing Lists

Adding Elements to a List

Changing Elements of a List

Concatenating and Repeating Lists

Inserting Item(s)

Removing or Deleting Items from a List
Sorting Items on a List
Using the count() Method on Lists
Testing for Membership on a List
Using Built-in Functions with List
Len()
Max()
Min()
Sum()
Sorted()
List()
Enumerate()
List Comprehension
Step 10: Tuples
How to Create a Tuple
Accessing Tuple Elements
Indexing
Negative Indexing
Slicing a Tuple
Changing, Reassigning, and Deleting Tuples
Replacing a Tuple
Reassigning a Tuple
Deleting a Tuple
Tuple Membership Test
Python Tuple Methods
Count(x)
Index(x)
Built-in Functions with Tuples
Len()
Max()
Min()
Sorted()

Sum()
Tuple()
Enumerate()
Iterating through a Tuple
Tuples vs. Lists
Step 11: Sets
Creating a Set
Changing Elements on a Set
Removing Set Elements
Set Operations
Set Union
Set Intersection
Set Difference
Set Symmetric Difference
Set Membership Test
Using Built-in Functions with Set
Len()
Max()
Min()
Sorted()
Sum()
Enumerate()
Iterating Through Sets
Frozenset
Step 12: Dictionary
Accessing Elements on a Dictionary
Adding and Modifying Entries to a Dictionary
Removing or Deleting Elements from a Dictionary
The pop()method
The popitem() method
The clear() method
Other Python Dictionary Methods

Update(other)
Item() method
Values() method
Keys() method
Setdefault() method
Copy() method
The fromkeys() method
Dictionary Membership Test
Iterating Through a Dictionary
Using Built-in Functions with Dictionary
Lens()
Sorted()
Creating a Dictionary with the dict() function
Dictionary Comprehension
Step 13:Python Operators
Arithmetic Operators
Assignment Operators
Relational or Comparison Operators
Logical Operators
Identity Operators
Membership Operators
Bitwise Operators
Understanding the Base 2 Number System
Precedence of Operators
Step 14:Built-in Functions
The range() function
The input() Function
Password Verification Program
Using input() to add elements to a List
The print() Function
abs()
max()

min()
type()
Step 15: Conditional Statements
if statements
if…else statements
if…elif…else statements
nested if…elif…else statements
Step 16: Python Loops
The for Loop
For Loop with string:
For Loop with list
for loop with a tuple
Using for loop with the range() function
The While Loop
Break Statement
Continue Statement
Pass Statement
Looping Techniques
Infinite loops (while loop)
Loops with top condition (while loop)
Loops with middle condition
Loops with condition at the end
Step 17: User-Defined Functions
1. def keyword
2. function name
3. parameters
4. colon (:)
5. docstring
6. statement(s)
7. return statement
Calling a Function
Using functions to call another function

Program to Compute for Weighted Average
Anonymous Functions
Lambda functions with map()
Lambda functions with filter()
Recursive Functions
Scope and Lifetime of a Variable
Step 18: Python Modules
Importing a Module
Python’s Math Module
Displaying the Contents of a Module
Getting more information about a module and its function
The Random Module
Usage of Random Module
Random Functions
Universal Imports
Importing Several Modules at Once
Step 19: Date and Time
Formatted Time
Getting Monthly Calendar
The Time Module
The Calendar Module
calendar.firstweekday( )
calendar.isleap(year)
calendar.leapdays(y1, y2)
calendar.month(year, month, w=2, l=1)
calendar.monthcalendar(year, month)
calendar.monthrange(year, month)
calendar.prmonth(year, month, w=2, l=1)
calendar.setfirstweekday(weekday)
calendar.weekday(year, month, day)
Datetime
Step 20: Namespaces

Scope
Step 21: Classes and Object-Oriented Programming
Defining a Class
Creating an Object
The __init__() method
Instance Variables
Adding an attribute
Deleting Objects and Attributes
Modifying Variables within the Class
Inheritance
Multiple Inheritance
Multilevel Inheritance
Step 22: Python Iterators
Creating a Python Iterator
Step 23: Python Generators
Step 24: Files
The File Object Attributes
File Operations
The Open() function
Writing to a File
Closing a File
Opening, Writing to, and Closing a Text File
Reading a Python File
The readlines() method
Line by Line Reading of Text Files with the ‘while’ loop
Line by Line Reading of Text Files using an Iterator
The ‘with statement’
Appending Data to a File
Renaming a File
The rename() method
Deleting a File
Binary Files

File Methods
File.writer(str)
File.writelines(sequence)
File.readline(size)
File.readlines()
File Positions: file.tell() and file.seek
File.tell()
File.seek()
File.read(n)
File.truncate([size])
File.flush()
File.close()
File.isatty()
File.fileno()
Step 25: Handling Errors or Exceptions
Syntax Errors
Runtime Errors
Built-in Exceptions
Catching Exceptions
try and except
try…finally
Python Cheat Sheets
Variable Assignment
Accessing Variable Values
Python Operators
Arithmetic Operators
Assignment Operators
Relational Operators
Logical Operators
Identity Operators
Membership Operators
Bitwise Operators

Strings

Lists

Tuple

Dictionary

Sets

Loops

Conditional Statements

if…else

if…elif…else

Built-in Functions

User-Defined Functions

Classes

Files

Text File Opening Modes

Binary File Opening Mode

File Operations

Date and Time

Python Modules

Help!

Google

FAQ

IRC (Internet Relay Chat)

Other Best Selling Books You Might Like!

Conclusion